Miles J. Murdocca - Vincent P. Heuring

Rutgers University University of Colorado

Principios de arquitectura de computadoras

Miles J. Murdocca - Vincent P. Heuring

Rutgers University — University of Colorado

Principios de arquitectura de computadoras

TRADUCCCIÓN

Ing. Fernando Szklanny

Universidad de Buenos Aires

Universidad Nacional de La Matanza

Universidad de Morón

REVISIÓN TÉCNICA

Ing. Elio de María

Universidad Nacional de La Matanza

Universidad de Morón

Universidad de Palermo

Prentice Hall

Pearson Educación

Argentina • Bolivia • Brasil • Colombia • Costa Rica • Chile • Ecuador • El Salvador • España • Guatemala • Honduras • México • Nicaragua • Panamá • Paraguay • Perú • Puerto Rico • República Dominicana • Uruguay • Venezuela

Amsterdam • Harlow • Londres • Menlo Park • Munich • Nueva Delhi • Nueva Jersey • Nueva York • Ontario • París • Sidney • Singapur • Tokio • Toronto • Zurich

Datos de catalogación bibliográfica

004 MUR
Murdocca, Miles
Principios de arquitectura de computadoras / Miles Murdocca y Vincent Heuring - 1° ed. - Buenos Aires: Pearson Education, 2002.
584p.; 25,5 x 19,5cm
Traducción de: Fernando Szklanny
ISBN: 987-9460-69-3
I. Heuring, Vincent II. Título - 1. Computadoras - Arquitectura

Editor: Enrique Baumann
Diseño de Tapa: Diego Linares
Diseño de Interior: Adriana Martínez
Traducción: Fernando Szklanny
Corrección: María E. Walas
Producción: Marcela Mangarelli / Cristian Rodríguez Tabares

Traducido de: PRINCIPLES OF COMPUTER ARCHITECTURE, 1st Edition by MURDOCCA, MILES, HEURING, VICENT P., published by Pearson Education, Inc, publishing as PRENTICE HALL, INC, Copyright © 2000.
ISBN: 0-20-0143664-7

Edición en español publicada por:
Copyright © 2002 PEARSON EDUCATION S.A.
Av. Regimiento de Patricios 1959 (C1266AAF), Buenos Aires, Rep. Argentina

PRENTICE HALL Y PEARSON EDUCACIÓN son marcas de propiedad de PEARSON EDUCATION S.A.

ISBN: 987-9460-69-3

Primera edición: enero de 2002
Queda hecho el depósito que dispone la ley 11.723

Impreso en Brasil por RR Donnelley, en el mes de enero de 2002.
Rua Epiacaba 90 –Vila Arapuá–
04257.170 - São Paulo SP

Para Ellen, Alexandra y Nicole

Para Gretchen

Índice de contenidos

Prólogo XIX

1. Introducción 1
1.1 Planteo general 1
1.2 Una historia breve 1
1.3 El modelo von Neumann 4
1.4 El modelo de interconexión a través de bus 5
1.5 Niveles de máquina 6
1.5.1 Compatibilidad "hacia arriba" 7
1.5.2 Los niveles 7
1.6 Un sistema de computación típico 12
1.7 Organización de la obra 12
1.8 Estudio de un caso: ¿Qué le ocurrió a las supercomputadoras? 14
Resumen 17
Para lectura posterior 17
Problemas 18

2. Representación de la información 19
2.1 Introducción 19
2.2 Números de punto fijo 20
2.2.1 Rango y precisión en números de punto fijo 21
2.2.2 La ley asociativa del álgebra no siempre funciona en la computadora 21
2.2.3 Sistemas de numeración posicionales 22
2.2.4 Conversión entre sistemas 23
2.2.5 Una primera mirada a la aritmética de las computadoras 29
2.2.6 Números signados en formato de punto fijo 30
2.2.7 Decimal codificado en binario 34
2.3 Formato de representación en punto flotante 35
2.3.1 Rango y precisión en números de punto flotante 36
2.3.2 La normalización y el esquema de bits implícitos 37

2.3.3 Representación de números de punto flotante dentro de la computadora. Introducción 38
2.3.4 Errores en la representación de punto flotante 41
Ejemplo 45
2.3.5 La norma de representación IEEE 754 45
2.4 Estudio de un caso: una falla en un misil defensivo Patriot causada por una pérdida de precisión 48
2.5 Códigos alfanuméricos 50
2.5.1 El conjunto de caracteres ASCII 50
2.5.2 El conjunto de caracteres EBCDIC 52
2.5.3 El código UNICODE 52
Resumen 55
Para lectura posterior 55
Problemas 56

3. Aritmética 61
3.1 Introducción 61
3.2 Suma y resta en punto fijo 61
3.2.1 Suma y resta en la representación de complemento a dos 62
3.2.2 Implementación circuital de sumadores y restadores 64
3.2.3 Suma y resta en representación de complemento a uno 67
3.3 Producto y cociente en punto fijo 69
3.3.1 Multiplicación de números sin signo 69
3.3.2 División sin signo 71
3.3.3 Producto y cociente signados 73
3.4 Aritmética de punto flotante 74
3.4.1 Suma y resta en formato de punto flotante 74
3.4.2 Producto y cociente en formato de punto flotante 75
3.5 Aritmética de alto rendimiento 76
3.5.1 Suma de alto rendimiento 76
3.5.2 Producto de alto rendimiento 78
3.5.3 División de alto rendimiento 82
3.5.4 Cálculo residual 85
Ejemplo. Sumador de alto rendimiento para palabras largas 86
3.6 Estudio de un caso: calculadora que utiliza el sistema decimal codificado en binario 88
3.6.1 La calculadora HP9100A 88
3.6.2 Suma y resta de números codificados en BCD 89
3.6.3 Suma y resta en BCD punto flotante 92
Resumen 94
Para lectura posterior 94
Problemas 95

4. La arquitectura de programación **99**
4.1 Componentes circuitales de la arquitectura de programación 100
4.1.1 Una revisión del modelo de bus 100
4.1.2 Memoria 101
4.1.3 La unidad central de proceso (CPU) 104
4.2 ARC, una computadora RISC 108
4.2.1 Memoria en ARC 109
4.2.2 El conjunto de instrucciones ARC 109
4.2.3 Formato del lenguaje simbólico de ARC 112
4.2.4 Formatos de instrucción en ARC 113
4.2.5 Formatos de datos en ARC 115
4.2.6 Descripción de las instrucciones de ARC 117
4.3 Directivas (seudo operaciones) 121
4.4 Ejemplos de programación en lenguaje ensamblador 122
4.4.1 Variantes en las arquitecturas y en los direccionamientos 125
4.4.2 Eficiencia de las arquitecturas de programación 128
4.5 El acceso a la información en la memoria. Modos de direccionamiento 129
4.6 Acceso a subrutinas y pilas 130
4.7 Entrada y salida en lenguaje simbólico 136
4.8 Estudio de un caso: la arquitectura de programación de la máquina virtual Java 139
Resumen 144
Para lectura posterior 144
Problemas 145

5. Los lenguajes y la máquina **151**
5.1 El proceso de compilación 151
5.1.1 Los pasos de la compilación 152
5.1.2 La especificación del mapeo 153
5.1.3 Cómo convierte el compilador los tres tipos de instrucciones al código ensamblador 153
5.1.4 Movimiento de datos 155
5.1.5 Instrucciones aritméticas 157
5.1.6 Control de secuencia 158
5.2 El proceso de ensamblado 160
5.2.1 El proceso de ensamblado y los ensambladores de dos pasadas 161
5.2.2 El proceso de ensamblado y la tabla de símbolos 164
5.2.3 Tareas finales del programa ensamblador 166
5.2.4 Ubicación de programas en memoria 168
5.3 Enlace y carga 168
5.3.1 Enlace (linking) 169
5.3.2 Carga 171

Ejemplo de programación 173
5.4 Macroinstrucciones (Macros) 175
5.5 Estudio de un caso: extensiones al juego de instrucciones - Las instrucciones SIMD Intel MMX™ y Motorola AltiVec™ 177
5.5.1 Fundamentos 178
5.5.2 Las arquitecturas básicas 178
5.5.3 Registros vectoriales 180
5.5.4 Operaciones de aritmética vectorial 182
5.5.5 Operaciones de comparación de vectores 184
5.5.6 Conclusiones de los casos de estudio 185
Resumen 185
Para lectura posterior 186
Problemas 187

6. Trayecto de datos y control 191
6.1 Fundamentos de la microarquitectura 192
6.2 Una microarquitectura para ARC 193
6.2.1 El trayecto de datos 193
6.2.2 La sección de control 201
6.2.3 Temporización 205
6.2.4 El desarrollo del microprograma 205
Ejemplo 214
6.2.5 Traps e interrupciones 216
6.2.6 Nanoprogramación 218
6.3 Control cableado 219
Ejemplo 227
6.4 Estudio de un caso: el lenguaje de descripción de hardware VHDL 227
6.4.1 Antecedentes 228
6.4.2 ¿Qué es VHDL? 228
6.4.3 La función mayoría y su descripción en lenguaje VHDL 229
6.4.4 Sistema lógico de nueve valores 233
Resumen 234
Para lectura posterior 234
Problemas 235

7. Memoria 243
7.1 Las jerarquías de la memoria 243
7.2 Memoria de acceso aleatorio 245
7.3 Organización de un circuito integrado 246
7.3.1 Construcción de una memoria grande a partir de memorias pequeñas 250
7.4 Módulos comerciales de memoria 251

7.5 Memoria de lectura 252
7.6 Memoria cache 254
7.6.1 Memoria cache de asignación asociativa 256
7.6.2 Memoria cache de asignación directa 260
7.6.3 Asignación asociativa por conjuntos de la memoria cache 262
7.6.4 Rendimiento de la memoria cache 263
7.6.5 Tasas de acierto y tiempos de acceso 265
7.6.6 Memorias cache multinivel 267
7.6.7 Administración de la memoria cache 268
7.7 Memoria virtual 270
7.7.1 Superposiciones (overlays) 270
7.7.2 Paginación 271
7.7.3 Segmentación 274
7.7.4 Fragmentación 276
7.7.5 Memoria virtual versus memoria cache 277
7.7.6 El buffer de traducción anticipada (Translation Lookaside Buffer - TLB) 278
7.8 Temas avanzados 279
7.8.1 Árboles decodificadores 279
7.8.2 Decodificadores para memorias de acceso aleatorio muy grandes 280
7.8.3 Memorias de direccionamiento por contenido (asociativas) 281
Ejemplo de diseño de una memoria: memoria RAM de doble puerto 285
7.9 Estudio de un caso: la memoria Rambus 285
7.10 Estudio de un caso: el sistema de memoria Pentium de Intel 289
Resumen 292
Para lectura posterior 292
Problemas 293

8. Entrada y salida 299
8.1 Arquitecturas de un único bus 300
8.1.1 Estructura de bus, protocolos y control 301
8.1.2 Frecuencias de reloj en el bus 302
8.1.3 El bus sincrónico 302
8.1.4 El bus asincrónico 303
8.1.5 Arbitraje del bus. Maestros y esclavos 305
8.2 Arquitecturas de bus basadas en puentes 307
8.3 Metodologías de comunicación 309
8.3.1 Entrada-salida programada 309
8.3.2 Entrada-salida administrada por interrupciones 310
8.3.3 Acceso directo a memoria 312
8.4 Estudio de un caso: comunicaciones en la arquitectura Pentium de Intel 314
8.4.1 El reloj del sistema, el reloj del bus y las velocidades del bus 314

8.4.2 Direcciones, datos, memoria y entrada-salida 315
8.4.3 Las palabras de datos no requieren alineación forzada 315
8.4.4 Ciclos de bus en la familia Pentium 315
8.4.5 Ciclos de bus de lectura y escritura de memoria 316
8.4.6 El ciclo de bus de lectura por bloques (burst) 317
8.4.7 Retención del bus para admitir pedidos de bus por parte de otro maestro 318
8.4.8 Velocidades de transferencia de datos 319
8.5 Almacenamiento masivo 320
8.5.1 Discos magnéticos 320
8.5.2 Cintas magnéticas 328
8.5.3 Tambores magnéticos 329
8.5.4 Discos ópticos 330
Ejemplo: tiempo de transferencia de un disco rígido 332
8.6 Dispositivos de entrada 333
8.6.1 Teclados 334
8.6.2 Tabletas digitalizadoras 335
8.6.3 Ratones y trackballs 335
8.6.4 Lápices ópticos y pantallas sensibles al tacto 336
8.6.5 Palancas de control (joysticks) 337
8.7 Dispositivos de salida 338
8.7.1 Impresoras láser 338
8.7.2 Pantallas de video 339
Resumen 342
Para lectura posterior 342
Problemas 343

9. Comunicaciones 347
9.1 Módems 347
9.2 Medios de transmisión 350
9.2.1 Líneas bipolares abiertas 351
9.2.2 Líneas de par trenzado 351
9.2.3 Cable coaxial 352
9.2.4 Fibra óptica 352
9.2.5 Satélites 352
9.2.6 Microondas terrestres 353
9.2.7 Radio 353
9.3 Arquitectura de redes: redes de área local 354
9.3.1 El modelo OSI 354
9.3.2 Topologías 356
9.3.3 Transmisión de datos 358

9.3.4 Puentes (bridges), encaminadores (routers) y puentes de enlace (gateways) 359
9.4 Errores de comunicación y códigos correctores de errores 360
9.4.1 Tasa de error 360
9.4.2 Detección y corrección de errores 361
9.4.3 Control de redundancia vertical 367
9.4.4 Control de redundancia cíclica 368
Ejemplo: corrección de dos errores 370
9.5 Arquitectura de redes: Internet 371
9.5.1 El modelo Internet 371
9.5.2 Revisión de puentes, encaminadores y conmutadores 376
9.6 Estudio de un caso: modo asincrónico de transferencia 378
9.6.1 Transferencia sincrónica contra transferencia asincrónica 379
9.6.2 ¿Qué es el modo asincrónico de transferencia? 380
9.6.3 Arquitectura de una red ATM 381
9.6.4 Perspectivas de ATM 383
Resumen 383
Para lectura posterior 383
Problemas 384

10. Avances en la arquitectura de computadoras 387
10.1 Análisis cuantitativo de la ejecución de un programa 388
10.1.1 Análisis cuantitativo del rendimiento 390
Ejemplo: cálculo del aumento de velocidad en un nuevo conjunto de instrucciones 391
10.2 De CISC a RISC 391
10.3 Segmentación del trayecto de datos 393
10.3.1 Instrucciones aritméticas, de salto y de acceso a memoria 393
10.3.2 Segmentación de las instrucciones 395
10.3.3 Cómo mantener ocupada la estructura 395
Ejemplo: análisis de la eficiencia de una estructura segmentada 397
10.4 Superposición de ventanas de registros 398
Ejemplo: código compilado para ventanas superpuestas y saltos retardados 401
10.5 Máquinas con instrucciones múltiples (superescalares). El PowerPC 601 406
10.6 Estudio de un caso: el procesador PowerPC como arquitectura superescalar 407
10.6.1 La arquitectura de programación del PowerPC 601 407
10.6.2 La arquitectura de hardware del PowerPC 601 407
10.7 Máquinas con palabra muy larga de instrucción 410
10.8 Estudio de un caso: la arquitectura Intel IA-64 (Merced) 410
10.8.1 Antecedentes: la arquitectura CISC 80x86 410
10.8.2 El procesador Merced: una arquitectura EPIC 411

10.9 Arquitecturas paralelo 414
10.9.1 La taxonomía de Flynn 416
10.9.2 Redes de interconexión 418
Ejemplo: red estrictamente no bloqueante 422
10.9.3 Asignación de un algoritmo a una arquitectura paralelo 424
10.9.4 Paralelismo de grano fino. La arquitectura CM-1 429
10.9.5 Paralelismo de grano grueso: CM-5 431
10.10 Estudio de un caso: el procesamiento paralelo en Sega Génesis 433
10.10.1 La arquitectura Sega Génesis 434
10.10.2 Operación de Sega Génesis 435
10.10.3 Programación de Sega Génesis 436
Resumen 437
Para lectura posterior 437
Problemas 439

A. Lógica digital 441
A.1 Introducción 441
A.2 Lógica combinatoria 441
A.3 Tablas de verdad 442
A.4 Compuertas lógicas 444
A.4.1 Implementación electrónica de la lógica 447
A.4.2 Buffers de tres estados 449
A.5 Propiedades del álgebra de Boole 450
A.6 Representación en suma de productos y diagramas lógicos 453
A.7 La forma producto de sumas 455
A.8 Lógica positiva o lógica negativa 456
A.9 La hoja de datos 458
A.10 Componentes digitales 460
A.10.1 Niveles de integración 461
A.10.2 Multiplexores 461
A.10.3 Demultiplexores 463
A.10.4 Decodificadores 464
A.10.5 Codificadores de prioridad 465
A.10.6 Matrices lógicas programables 466
Ejemplo: un sumador con arrastre serie 469
A.11 Lógica secuencial 471
A.11.1 El circuito biestable (flip flop) S-R 472
A.11.2 El flip flop S-R sincrónico 474
A.11.3 El flip flop D y la configuración maestro-esclavo 476
A.11.4 Flip flops J-K y T 477
A.12 Diseño de máquinas de estado 479

Ejemplo: un detector de secuencia 482
Ejemplo: un controlador para una máquina expendedora 485
A.13 El modelo Mealy y el modelo Moore 487
A.14 Registros 488
A.15 Contadores 489
Resumen 491
Para lectura posterior 491
Problemas 492

B. Simplificación de circuitos lógicos 499
B.1 Reducción de lógica combinatoria y de lógica secuencial 499
B.2 Reducción de las expresiones de dos niveles 499
B.2.1 El método algebraico 500
B.2.2 El método de los mapas K 501
B.2.3 El método tabular 509
B.2.4 Simplificación lógica: su efecto sobre la velocidad y la eficiencia 517
B.3 Reducción de estados 521
B.3.1 El problema de la asignación de estados 524
Ejemplo de reducción: un detector de secuencia 526
B.3.2 Tablas de transiciones 528
Ejemplo de tabla de transiciones: un detector de mayoría 533
Resumen 535
Para lectura posterior 535
Problemas 536

Índice analítico 541

Prólogo

Acerca del libro

Nuestro propósito al escribir este texto es describir el funcionamiento interno de la computadora digital moderna a un nivel que logre desmitificar lo que ocurre dentro de esta. El único requisito previo para la lectura de *Principios de arquitectura de computadoras* es poseer un adecuado conocimiento práctico de un lenguaje de programación de alto nivel. La selección del material cubre temas que habitualmente forman parte de un primer curso de arquitectura de computadoras o de organización de computadoras. La profundidad y la amplitud con que se han tratado los temas tienen como propósito colocar al estudiante principiante en un camino sólido para la continuidad de sus estudios en disciplinas relacionadas con las computadoras.

Cuando se requiere crear un texto sobre arquitectura de computadoras, las soluciones técnicas surgen en forma natural, en tanto que las características organizativas se relacionan con las prestaciones más importantes. Entre las características que recibieron mayor atención a lo largo de este texto se incluyen la elección de la arquitectura de programación (ISA, *Instruction Set Architecture*), el uso de los casos de estudio y un importante desarrollo de ejemplos y ejercicios.

La arquitectura de programación

Los textos que analizan los lenguajes absolutos de programación tienen dos criterios para el tipo de arquitectura de instrucciones a utilizar: o se adopta un modelo propio, o se elige uno de los tantos modelos comerciales existentes en el mercado. La elección afecta al instructor, que puede necesitar una arquitectura compatible con una plataforma de la que dispone para los trabajos de programación que le asigna a sus estudiantes. Para mayor complicación, la plataforma local puede cambiar de semestre en semestre: ayer pudo ser MIPS, hoy un Pentium, mañana un SPARC. Los autores optaron por ambos caminos al adoptar como arquitectura la de programación un subconjunto de la arquitectura *SPARC*, a la que llamaron ARC (*A RISC Computer*, una computadora RISC). Esta se utiliza a lo largo del texto, siendo complementada con herramientas de programación in-

dependientes de la plataforma, que permiten simular tanto la arquitectura ARC como las arquitecturas que corresponden a los procesadores Intel X86 (Pentium) o MIPS.

Casos de estudio, ejemplos y ejercicios

Cada capítulo contiene al menos un caso de estudio como elemento que le permite al alumno acercarse a ejemplos reales sobre el tema. Esto ubica al tema en perspectiva, lo que, a juicio de los autores, aporta al material un aire de realismo e interés.

Se han incorporado tantos ejemplos y ejercicios como ha sido posible para cubrir los puntos más significativos del texto. En el sitio web que acompaña al texto se encuentran disponibles ejemplos adicionales y soluciones a los problemas.

Cobertura de los temas

El enfoque de este libro presenta a la computadora como un sistema integrado. Si se debiera elegir un subtítulo, este podría ser "Un enfoque integrado", lo que reflejaría las relaciones de alto nivel que vinculan a los distintos elementos entre ellos. Cada tema se cubre en el contexto de la integridad de la máquina de la que es parte, y con la perspectiva de determinar cómo afecta la implementación a la conducta del sistema. Por ejemplo, la precisión finita de los números binarios se relaciona con la cantidad de unos que pueden sumarse a un número expresado en punto flotante, antes de que el error en la representación exceda el valor de 1. (Por esta razón, debe evitarse el uso de números de punto flotante como variables de control en rutinas tipo lazo.) Otro ejemplo es el tema de la vinculación entre subrutinas, que se trata con la expectativa de que el lector pueda enfrentarse alguna vez con la necesidad de programar en el lenguaje C o con programas escritos en Java que invocan subrutinas escritas en otros lenguajes de alto nivel, como Fortran.

Como otro ejemplo del enfoque integrado, los conceptos de detección y corrección de errores se analizan en el contexto de la transmisión y almacenamiento masivos, ante la perspectiva de que el lector pueda encarar aplicaciones referidas a redes (en las que los errores y las pérdidas de datos son un hecho cotidiano) o deba lidiar con elementos de almacenamiento poco confiables, como los CD-ROM (*compact disk-read only memory*).

La arquitectura de las computadoras impacta sobre la mayor parte de las actividades que los profesionales de la computación realizan diariamente, por lo que el enfoque integrado orienta el texto hacia la diversidad de áreas afines en las que dicho profesional debe capacitarse. Este énfasis es consecuencia de una transición que está ocurriendo en muchos planes de estudio de carreras de grado relacionadas con la computación. A medida que las arquitecturas de las computadoras se complican cada vez más, deben ser tratadas con niveles de abstracción cada vez mayores, ya que, de alguna manera, se han

vuelto dependientes de la tecnología. Por esta razón, la mayor parte del texto trata sobre la arquitectura de computadoras con un enfoque de alto nivel, en tanto que los aspectos de menor nivel, dependientes de lo tecnológico, se tratan en los apéndices y en los casos de estudio.

Los capítulos

En el capítulo 1, "Introducción", se da comienzo al texto con una breve historia de la arquitectura de las computadoras, avanzando a través de las partes básicas que las componen, con lo que el estudiante termina con una visión de los sistemas de computación desde el alto nivel. Se presenta el modelo convencional de Von Neumann, seguido por el análisis descriptivo de una computadora típica. Este capítulo se propone sentar las bases para permitir un análisis más profundo en capítulos sucesivos.

En el capítulo 2, "Representación de la información", se cubren los temas básicos relacionados con la representación de datos. Se analizan las representaciones de números signados, a través de las convenciones de complemento a uno, complemento a dos, magnitud y signo, y representación en exceso. Este capítulo también toma en cuenta la representación decimal codificada en binario (BCD), utilizada habitualmente en calculadoras. Además, se cubre la representación de números en formato punto flotante, incluyendo la norma IEEE 754 de uso común en números binarios. Por ultimo, también se analizan las representaciones ASCII, EBCDIC y Unicode para caracteres.

En el capítulo 3, "Aritmética", se analizan la aritmética de computadoras y las representaciones avanzadas. Se estudian los procedimientos para realizar operaciones de suma, resta, producto y cociente en notación de punto fijo. Se consideran también las representaciones de complemento a nueve y complemento a diez, utilizadas habitualmente en el cálculo BCD, así como las operaciones aritméticas en BCD y en punto flotante. Se desarrollan, además, los métodos requeridos para mejorar la eficiencia de estas operaciones, como la suma con arrastre anticipado, la multiplicación matricial y la división a través de iteración funcional. Para completar, se analiza brevemente la teoría de residuos, para introducir un enfoque poco convencional de alto rendimiento.

En el capítulo 4, "La arquitectura de programación", se introducen los conceptos sobre los componentes arquitectónicos básicos requeridos para la ejecución de los programas. Se analizan los conceptos de lenguaje de máquina y de ciclos búsqueda-ejecución. Se detalla la organización de la unidad central de proceso, se analizan las funciones del bus del sistema en la articulación entre las unidades aritmético-lógica, de memoria, de entrada y salida, y se detalla la estructura de la unidad de control.

En el contexto de la arquitectura ARC (*A RISC Computer*), ligeramente basada en la arquitectura comercial SPARC, se analiza el tema de la programación en lenguaje de

máquina. La arquitectura descriptiva ARC mantiene, con respecto a SPARC, los nombres y formatos de las instrucciones, los formatos de los datos y la sintaxis de su lenguaje simbólico de programación, pero, por otra parte, presenta una gran simplificación respecto de aquella. En la mayor parte del capítulo se utilizan solo 15 de las instrucciones de SPARC, admitiendo en principio un único formato para datos, de 32 bits sin signo. Se analizan los formatos de instrucciones, así como los modos de direccionamiento. Asimismo se trata con distintos enfoques el manejo de subrutinas, que incluye un análisis detallado de la transferencia de parámetros mediante el uso de una pila.

En el capítulo 5, "Los lenguajes y la máquina", se relaciona el enfoque del programador de un sistema de computación con la arquitectura de la máquina. Se plantean temas de programación del sistema, con el objetivo de colocar el lenguaje de bajo nivel a la vista del programador. El capítulo comienza con una explicación del proceso de compilación, para lo cual primero se analizan los pasos involucrados en la compilación y luego se plantea la generación del código objeto. Se describe el proceso de traducción a lenguaje de máquina a través de un ensamblador (***assembler***) de dos etapas, agregando a la descripción ejemplos de la generación de tablas de símbolos. Se cubren también los conceptos asociados con macros y con los procesos de vinculación (***linking***) y de carga.

En el capítulo 6, "Trayecto de datos y control", se realiza el análisis paso a paso de una unidad de control y los caminos de los datos. Se estudian los dos métodos comúnmente utilizados para las unidades de control: el del control microprogramado y el de la unidad de control cableada. El instructor podrá adoptar un método y omitir el otro, o cubrir ambos en función de su disponibilidad de tiempo. Las unidades de control microprogramadas y las unidades cableadas, utilizadas como ejemplos, implementan el subconjunto ARC del lenguaje de programación SPARC presentado en el capítulo 4.

En el capítulo 7, "Memoria", se analiza la memoria de la computadora, comenzando por la organización de una memoria de acceso al azar básica, y continuando con conceptos avanzados como memoria virtual y ***cache***. Se analizan los esquemas tradicionales de mapeo de memoria *cache*, tales como el directo y el asociativo, así como el *cache* en varios niveles. También se tratan temas como el paginado (***overlays***), las políticas de reemplazo, la segmentación, la fragmentación y el concepto de ***buffer*** **de traducción anticipada** (TLB, *translation lookaside buffer*).

En el capítulo 8, "Entrada y salida", se cubren los métodos de acceso y comunicaciones en buses. Se analiza también la vinculación entre buses. Además, trata sobre los distintos dispositivos de entrada-salida de uso habitual, como discos, teclados, impresoras y pantallas.

En el capítulo 9, "Comunicaciones", se analizan las arquitecturas de redes, con enfoques sobre módems, redes de área local (**LAN**, *local area network*) y redes de área extendida (**WAN**, *wide area network*). Se plantea especial interés en el análisis de las arquitecturas de redes, analizando en forma sencilla los protocolos que enfocan las prestaciones clave de dichas arquitecturas de redes. Se analizan en profundidad la detección y la corrección de errores, y se introduce, en el contexto de Internet, el esquema de protocolo TCP/IP.

En el capítulo 10, "Avances en la arquitectura de computadoras", se cubren las características avanzadas de las arquitecturas que han aparecido o cambiado de enfoque en los últimos años. La primera parte del capítulo analiza los conceptos de la arquitectura de procesadores para las computadoras con conjunto reducido de instrucciones, más conocidas como de arquitectura RISC (*Reduced Instruction Set Computer*) y las consecuencias derivadas de esta arquitectura. La última parte de este capítulo trata acerca de las máquinas de instrucción multiple y de palabra muy larga de instrucción (**VLIW, *Very Large Instruction Word***). A través de un caso de estudio se hacen visibles al programador las prestaciones de la arquitectura RISC, para lo cual se realiza el análisis paso a paso de un programa SPARC generado por un compilador C, con explicaciones acerca del uso de registros, de pilas y de segmentación (***pipelining***). El capítulo cubre también las arquitecturas distribuidas y paralelas, así como las redes de interconexión utilizadas en el procesamiento paralelo y en el procesamiento distribuido.

En el apéndice A, "Lógica digital", se analizan los temas asociados con lógica combinatoria y secuencial, además de establecer la base que permita entender el diseño lógico de los componentes analizados en el resto del texto. El apéndice A se inicia con una descripción de tablas de verdad, álgebra de Boole y ecuaciones lógicas. Asimismo, se describe la síntesis de circuitos lógicos combinatorios, explorándose una gran cantidad de ejemplos. Se analizan los componentes de mediana escala de integración (**MSI, *Medium Scale Integration***), tales como multiplexores y decodificadores; se completa el análisis con ejemplos de diseño de circuitos que utilizan componentes MSI.

En este apéndice se analiza también el concepto de lógica sincrónica, iniciándose con una introducción de los aspectos de temporización de los circuitos biestables (*flip-flops*). Se trata, asimismo, la síntesis de circuitos lógicos sincrónicos, con relación a sus diagramas de estados, sus tablas de estados y el diseño de lógica sincrónica.

En el apéndice B, "Simplificación de circuitos lógicos", que es un complemento del apéndice A, se cubren los métodos de reducción de lógica tanto combinatoria como secuencial. Se analizan los métodos de minimización basados en la simplificación algebraica, así como en los diagramas de Karnaugh y los métodos tabulares de Quine-McCluskey para funciones simples y múltiples. También se analizan la reducción y la asignación de estados.

Ordenamiento de los capítulos

El orden de los capítulos ha sido planteado de modo tal que los mismos puedan desarrollarse en orden numérico, aun cuando el instructor puede modificar el ordenamiento propuesto para adecuarlo a un programa o plan de estudios particular. La figura P.1 plantea los requisitos necesarios para establecer relaciones entre los capítulos, detallándose a continuación algunas consideraciones especiales respecto de la secuencia planteada.

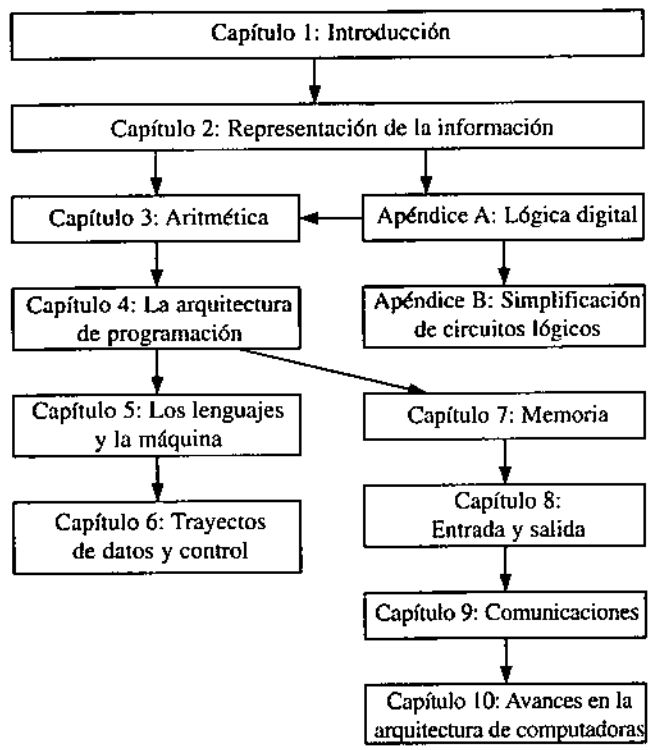


Figura P.1 • Requerimientos de correlatividad entre capítulos.

El capítulo 2, "Representación de la información", debería desarrollarse antes que el capítulo 3, "Aritmética", dado que este requiere de los conocimientos de representación de datos. Los apéndices A, "Lógica digital", y B, "Simplificación de circuitos lógicos", pueden omitirse si el tema de lógica digital se ha desarrollado en una asignatura previa de la carrera; pero si el material no se cubre, las estructuras de algunos componentes (tales como una unidad aritmético-lógica o un registro) pueden quedar en el misterio si no se desarrolla al menos el apéndice A antes que el capítulo 3.

Los capítulos 4, "La arquitectura de programación", y 5, "Los lenguajes y la máquina", aparecen en la primera mitad del texto por dos razones: (1) introducen al lector en el funcionamiento de una computadora a un nivel relativamente alto, el que permite un enfoque desde arriba hacia abajo (*top-down*) del estudio de la arquitectura de la computadora, y (2)

es importante iniciar la programación en lenguaje de bajo nivel en forma temprana solo si se realizan prácticas de programación como parte del curso.

El capítulo 10, "Avances en la arquitectura de computadoras", aparece generalmente en los niveles de grado de los cursos de arquitectura; por ende, debería desarrollarse solo si el tiempo lo permite, luego de haber desarrollado todo el contenido de los capítulos anteriores.

El sitio web

Acompañando a este texto, se ofrece un sitio web vinculado con el mismo:

```
http://www.pearsonedlatino.com/murdocca
```

Este sitio contiene un conjunto de elementos de apoyo, como software y diapositivas en formato Acrobat, y una fe de erratas. Para quienes adopten el libro como texto, también se encuentran disponibles las soluciones a los problemas, así como problemas tipo para exámenes con sus respectivas soluciones (aquellos docentes que deseen acceder a esta información, deberán conectarse con su representante de Prentice Hall).

Herramientas de software

Se provee un ensamblador y un simulador para la arquitectura ARC, así como subconjuntos de los lenguajes de máquina de los procesadores MIPS y X86 (Pentium). Escritos como aplicaciones Java para aprovechar su portabilidad, los ensambladores y simuladores pueden descargarse desde el sitio web mencionado.

Trasparencias

Todas las figuras y tablas de *Principios de arquitectura de computadoras* se han incluido en una presentación de diapositivas bajo Adobe Acrobat. La versión gratuita de Acrobat Reader se puede obtener en www.adobe.com.

Problemas de exámenes y soluciones

Los problemas tipo y sus soluciones han sido ampliamente probados en clase. Los problemas son los suficientes en cantidad como para servir en exámenes y evaluaciones, y para ser utilizados como problemas prácticos en clases de repaso a criterio del instructor. Los problemas tipo (que han sido incluidos con sus soluciones) y las soluciones a los problemas se encuentran a disposición del instructor que adopte el texto. (Para acceder a este sector del sitio web, deberá ponerse en contacto con su representante de Prentice Hall. Los autores solo requieren que este material no sea colocado en algún otro sitio web.)

Si surgieran errores

A pesar de los mejores esfuerzos de los autores, editores, revisores, evaluadores y traductores, este libro seguramente contiene errores. Antes de informar acerca de algún error, sírvase verificar en www.pearsonedlatino.com/murdocca para ver si ya ha sido catalogado. Los errores detectados deberán ser informados a pocabugs@cs.rutgers.edu. El asunto del mensaje de correo electrónico remitido a esa dirección deberá indicar el numero del capítulo en el que se ha encontrado el error.

Créditos y agradecimientos

Los autores no han trabajado solos durante la creación de este libro, por lo que agradecen profundamente el apoyo de la gran cantidad de gente que ha influido en la preparación de este texto y en el pensamiento de los autores en general. Los autores desean agradecer, primeramente, a los editores Paul Becker, Thomas Robbins y Eric Frank, quienes tuvieron el panorama y la visión para guiar este libro y sus materiales de apoyo a lo largo de todo el trayecto que llevó a su concreción. Donald Chiarulli tuvo una influencia importante sobre una edición temprana de este libro, el que fue probado en clase en las universidades de Rutgers y de Pittsburgh. Saul Levy, Donald Smith, Vidyadhar Phalke, Ajay Bakre, Jinsong Huang y Srimat Chakradhar colaboraron con la evaluación del material en Rutgers y aportaron parte del texto, problemas y algunas aclaraciones de alto valor. Brian Davison y Shridhar Venkatanarisam trabajaron sobre la primera versión de las soluciones y aportaron muchos comentarios de gran ayuda. Irving Rabinowitz aportó una buena cantidad de problemas. Larry Greenfield hizo su aporte desde el punto de vista de un estudiante nuevo en el tema, por lo que se debe tomar en cuenta su participación en la organización del capítulo 2. También quieren destacar su reconocimiento a Blair Gabett Bizjak por haber provisto el esqueleto para la mayor parte del material referido a redes de área local. Ann Yasuhara aportó el texto sobre las contribuciones de Alan Turing a la ciencia de la computación. William Waite, de la Universidad de Colorado, Boulder, proporcionó buena cantidad de los ejemplos en lenguaje de máquina. Ann Root, de la Universidad de Colorado, Boulder, realizó un trabajo estupendo en el desarrollo de las herramientas de simulación que se encuentran disponibles en el sitio web. La población estudiantil de las universidades de Rutgers y Colorado sirvió como amplio campo de ensayo del material, por lo que los autores desean reconocer su paciencia y sus recomendaciones durante la elaboración del material para el libro.

También desean hacer público su agradecimiento a los revisores, quienes colaboraron ampliamente al sugerir cambios a los manuscritos, lo que incluye a Perry Alexander (University of Cincinnati), Thomas L. Casavant (University of Iowa, Electrical and Computer Engineering Department), J. Kelly Flanagan (Brigham Young University, Dept. of Computer Science), Shelly Dalton Goggin (University of Colorado, Denver),

Nikrouz Faroughi (California State University Sacramento, Computer Science Dept.), Mark Jones (Virginia Tech., Department of Electrical and Computer Engineering), Thuy T. Le (Profesor Adjunto, Electrical Engineering Dept., San Jose State University, e Investigador Senior, High Performance Computing Group, Fujitsu America, Inc.), Gary Lippman, (Profesor, California State University Hayward, Dept. of Mathematics and Computer Science), Walid Najjar (Colorado State University, Computer Science Dept.), William Hsu (San Francisco State University), Jyh-Charn (Steven) Liu (Computer Science Dept., Texas A&M University), David B. Wortman (University of Toronto, Dept. of Computer Science), Janusz Zalerwski (University of Central Florida, Orlando, Florida, Dept. of Electrical and Computer Engineering).

Yo, Miles J. Murdocca quiero expresar mi reconocimiento a mis padres Dolores y Nicolas Murdocca, mi hermana Marybeth y mi hermano Mark por su aliento. Mi esposa Ellen y mis hijas Alexandra y Nicole han sido una fuente interminable de aliento e inspiración. No creo haber podido encontrar la energía necesaria para acometer el esfuerzo realizado sin todo el apoyo recibido de ellos.

Yo, Vincent P. Heuring, deseo agradecer el apoyo de mi esposa Gretchen, quien fue excesivamente paciente y alentadora durante todo el proceso de escritura de este libro.

Existen, seguramente, más personas e instituciones que han colaborado con este libro, tanto en forma directa como indirecta, cuyos nombres pueden haber sido omitidos en forma accidental. A esa gente y a esas instituciones, los autores quieren ofrecer aquí su tácito reconocimiento y sus disculpas por la omisión de los reconocimientos explícitos que hubiesen correspondido.

Miles J. Murdocca
Rutgers University
murdocca@cs.rutgers.edu

Vincent P. Heuring
University of Colorado - Boulder
heuring@colorado.edu

Capítulo 1

Introducción

1.1 Planteo general

El concepto de **arquitectura de computadoras**, enfocado desde el punto de vista de un programador, se relaciona con el comportamiento funcional de un sistema de computación. Este punto de vista incluye aspectos tales como el tamaño de los diferentes tipos de datos (por ejemplo, el uso de 16 bits para la representación de enteros) y los tipos de operaciones que se pueden realizar (como la suma, la resta y los llamados a subrutinas). La **organización de computadoras** se refiere a las relaciones estructurales que no son visibles para el programador, como las interfaces hacia los dispositivos periféricos, la frecuencia del reloj y la tecnología utilizada en las memorias. Este texto trata tanto de la arquitectura como de la organización, haciendo uso del término "arquitectura" para referirse en forma amplia tanto a la arquitectura como a la organización de la computadora.

En la arquitectura de computadoras suele emplearse el concepto de **niveles**. La idea básica es la de la existencia de muchos niveles, o enfoques, desde los cuales considerar a la computadora, los que van desde el nivel superior, en el que el usuario ejecuta programas o utiliza la computadora, al nivel inferior, consistente en transistores y cables. Entre los niveles superior e inferior existe una cantidad de niveles intermedios. Antes de analizar estos niveles presentaremos una breve historia de la computación, con el objeto de obtener un enfoque claro acerca de su evolución.

1.2 Una historia breve

La existencia de dispositivos mecánicos destinados al control de operaciones complejas se conoce al menos desde el siglo XVI, cuando ya en las cajas de música se utilizaban cilindros rotativos con clavijas en una forma similar a las de la actualidad. Las máquinas para la realización de cálculos, como ejemplo de elementos que no solo repiten una melodía predeterminada, aparecieron en el siglo siguiente.

Blas Pascal (1623-1662) desarrolló una calculadora mecánica con el objeto de colaborar en el cálculo de impuestos que realizaba su padre. La calculadora Pascalina, desa-

rrollada por Pascal, consta de ocho diales que se conectan a un tambor (véase la figura 1.1), vinculados entre ellos en una forma innovadora que provoca que uno de los diales se desplace en una posición cuando se produce un arrastre desde un dial de menor posición. Se coloca una ventana por encima del dial para permitir la observación de la posición, en forma similar a la del odómetro de un auto, con la diferencia de que los diales se encuentran colocados horizontalmente, a la manera del dial de un teléfono rotativo.

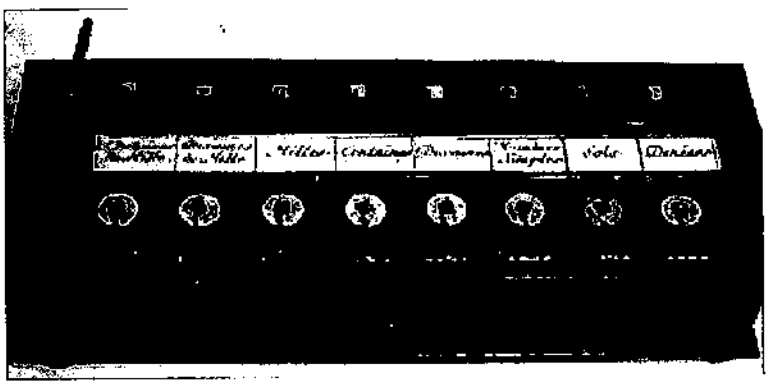


Figura 1.1 • La máquina calculadora de Pascal. (Reproducida con autorización del Departamento de Ciencias de la Computación, Universidad de Manchester.)

Aún existen algunas de las máquinas sumadoras de Pascal, de las que comenzó a construir en 1642. No obstante, recién alrededor del año 1800 hubo quien agrupara los conceptos de control y cálculo mecánicos en una única máquina que pudiese reconocerse hoy como formada por las partes básicas integrantes de una computadora. Ese alguien fue Charles Babbage.

A Charles Babbage (1791-1871) se lo menciona a veces como "el abuelo" de la computadora, más que como "el padre" de la misma, dado que nunca construyó una versión práctica de las máquinas que diseñó. Babbage vivió en Inglaterra en una época en la que las tablas matemáticas eran de uso común en la navegación y en el trabajo científico. Esas tablas se computaban manualmente y, por consiguiente, contenían gran número de errores. Frustrado por las imprecisiones, Babbage se decidió a crear una máquina que resolviera cálculos a través del posicionamiento de engranajes y de su movimiento. La máquina que diseñó permitía incluso la generación de una plancha para impresión, lo que eliminaba los errores que podrían producirse debido a la intervención del tipógrafo.

Las máquinas de Babbage poseían medios para la lectura de los datos de entrada, el almacenamiento de los datos, la producción de los datos de salida y el control automático de la operación de la máquina. Estas son funciones básicas que se encuentran en prácticamente cualquier computadora moderna. Babbage creó un pequeño prototipo de su **máquina diferencial**, la que evalúa polinomios utilizando el método de las diferencias finitas. El éxito del concepto de la máquina diferencial le permitió lograr apoyo gubernamental para la **máquina analítica**, mucho más grande y sofisticada, la que poseía

mecanismos para realizar **bifurcaciones** (toma de decisiones) y medios para programarla empleando tarjetas perforadas, en forma similar a lo que se conoce como **tejido por patrones Jacquard**.

Babbage diseñó la máquina analítica aunque nunca la construyó, dado que las tolerancias mecánicas requeridas por su diseño no podían lograrse con la tecnología de esa época. En 1991, el Museo de Ciencias de Londres construyó una versión de la máquina diferencial de Babbage, la que aún hoy puede ser observada.

Pasó más de un siglo, hasta el comienzo de la Segunda Guerra Mundial, antes de que surgiera un nuevo avance importante en la computación. En Inglaterra, los submarinos alemanes **U-boat** provocaban grandes daños en los barcos aliados. Los U-boats recibían comunicaciones desde sus bases ubicadas en Alemania, para las que se utilizaban códigos encriptados implementados por una máquina desarrollada por Siemens AG y conocida como **ENIGMA.**

El proceso de encriptado de la información ya se conocía desde mucho tiempo atrás, y hasta el presidente estadounidense Thomas Jefferson (1743-1826) había desarrollado un antecesor de ENIGMA, sin haber llegado a construir la máquina. El proceso de decodificar la información encriptada era una tarea mucho más pesada. Este fue el problema que orientó los esfuerzos de Alan Turing (1912-1954) y de otros científicos ingleses hacia la creación de sistemas para descifrar códigos. Durante la Segunda Guerra Mundial, Turing era el criptógrafo más importante de Inglaterra, siendo conocido como quien cambió el concepto de la criptografía, que pasó de ser tema para quienes se ocupaban de descifrar idiomas antiguos a ser un tema de y para matemáticos.

De Bletchley Park, Inglaterra, lugar de trabajo de Turing, surgió **Colossus**, una exitosa máquina decodificadora. Su funcionamiento se basaba en la alimentación de la máquina a través de una cinta de papel, cuyos datos se procesaban en válvulas de vacío, realizando los cálculos entre esas válvulas de vacío y una segunda cinta de papel con la que se alimentaba a la máquina. La programación se efectuaba a través de paneles de conexión. Se desconoce la participación de Turing en las distintas versiones de la máquina debido al secreto que rodeaba al proyecto, pero algunos aspectos de su vida y obra se pudieron ver en la obra *Breaking the Code*, que fuera puesta en escena en Londres y Nueva York sobre finales de la década de 1980.

Alrededor de la misma época en que Turing desarrollaba sus esfuerzos, J. Presper Eckert y John Mauchly comenzaban a crear una máquina que pudiera usarse para el cálculo de tablas de trayectorias balísticas, para la Marina de los Estados Unidos. El resultado de los esfuerzos de Eckert y Mauchly fue **ENIAC** (*Electronic Numerical Integrator and Computer*). ENIAC estaba constituida por 18.000 válvulas de vacío, las que conformaban la sección de cálculo de la máquina. La programación y el ingreso de los datos se realizaban a través de la fijación de elementos conmutadores (llaves) y del intercambio de cables. No existía el concepto de programa almacenado, ni tampoco una unidad de memoria central; no obstante, estas limitaciones no eran importantes dado que todo lo que ENIAC debía hacer era el cálculo de trayectorias balísticas. A pesar de no haber podido

entrar en servicio hasta 1946, luego del final de la guerra, fue considerada un modelo exitoso y utilizada durante nueve años.

Luego del éxito de ENIAC, Eckert y Mauchly, que trabajaban en la Escuela Moore en la Universidad de Pennsylvania, se vincularon con John von Neumann (1903-1957), quien desarrollaba actividades en el Instituto de Estudios Avanzados de Princeton. En forma conjunta trabajaron sobre el desarrollo de una computadora de programa almacenado a la que llamaron **EDVAC**. Debido al surgimiento de un conflicto, los grupos de Pennsylvania y Princeton se separaron. El concepto de la computadora con programa almacenado, no obstante, prosperó; el resultado fue un modelo funcional de computadora de programa almacenado, conocido como **EDSAC**, construido por Maurice Wilkes, de la Universidad de Cambridge, en 1947.

1.3 El modelo von Neumann

Las computadoras digitales convencionales presentan un aspecto común que se atribuye a von Neumann, aunque los historiadores coinciden en que el diseño en cuestión fue obra de todo el equipo. El **modelo von Neumann** consta de cinco componentes principales, tal como lo ilustra la figura 1.2. La **unidad de entrada** provee las instrucciones y los datos requeridos por el sistema, los que se almacenan en la **unidad de memoria**. Las instrucciones y los datos se procesan en la **unidad aritmético-lógica** (ALU) bajo la dirección de la **unidad de control**. Los resultados obtenidos se envían a la **unidad de salida**. El conjunto constituido por las unidades aritmético-lógica y de control se designa habitualmente bajo el nombre de **unidad central de proceso** (CPU). La mayoría de las computadoras comerciales pueden descomponerse en estas cinco unidades.

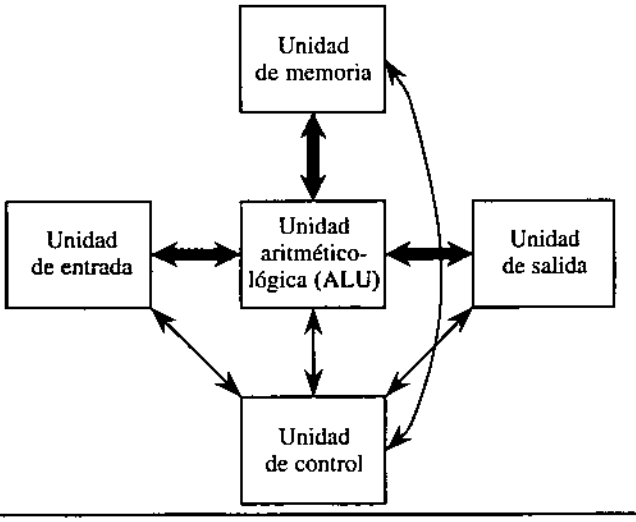


Figura 1.2 • Una computadora digital según el modelo von Neumann. Las flechas gruesas representan rutas de datos. Las flechas más finas representan rutas de control.

El **programa almacenado** es el aspecto más importante del modelo von Neumann. Los programas se almacenan en la memoria de la computadora junto con los datos a procesar. Si bien en la actualidad podemos plantear esto como un hecho concreto, antes del desarrollo de las computadoras de programa almacenado, los programas se almacenaban en un medio externo, tales como los tableros de clavijas –anteriormente mencionados–, cintas o tarjetas perforadas. En la computadora de programa almacenado, el programa puede manipularse como si se tratara de datos. Este concepto da origen a los compiladores y sistemas operativos, y es la base de la gran versatilidad de las computadoras modernas.

1.4 El modelo de interconexión a través de bus

Si bien el modelo von Neumann prevalece en la estructura de las actuales computadoras, el mismo ha sido modernizado. La figura 1.3 muestra el modelo de una computadora que utiliza el sistema de interconexión a través de lo que se denomina **bus del sistema**. El modelo considera que el sistema de computación está constituido por tres subconjuntos, la CPU, la memoria y la entrada-salida (E/S). Este refinamiento del modelo von Neumann combina la ALU y la unidad de control en un solo bloque funcional, la CPU. Las unidades de entrada y de salida se combinan, asimismo, en una única unidad de entrada-salida.

Lo más importante de este modelo es que realiza las comunicaciones entre los componentes por medio de un camino compartido conocido como **bus del sistema***, constituido a su vez por un **bus de datos** (que trasporta la información que se está trasmitiendo), un **bus de direcciones** (que determina hacia donde está siendo enviada dicha información) y un **bus de control** (que describe aspectos de la forma en que se está llevando a cabo la mencionada trasferencia de información). Existe también un **bus de alimentación**, que lleva energía eléctrica a los componentes. Este último no figura en los esquemas pero se sobreentiende su presencia. Algunas arquitecturas pueden tener, además de los anteriores, un **bus de entrada-salida**.

Físicamente, los buses están constituidos por conjuntos de cables agrupados de acuerdo con su función. Un bus de datos de 32 bits contiene 32 cables individuales, cada uno de los cuales trasporta un bit de datos (distinguiéndolo de la información de direcciones o control). En este sentido, el bus del sistema es, en realidad, un grupo de buses individuales clasificados de acuerdo con su función.

El bus de datos trasporta datos entre los componentes del sistema. Algunos sistemas tienen buses de datos separados para el ingreso o la salida de información hacia o desde

* *N. de T.*: Se ha preferido mantener la denominación original de **bus** para distinguir este conjunto de líneas eléctricas, a pesar del posible reemplazo por términos como barra, mazo o similares, utilizados para definir conjuntos de cables, debido a que dicho término ya es de uso común en el ambiente de la computación.

la CPU, en cuyo caso existirán un **bus de entrada de datos** y otro **bus de salida de datos**. Pero, más a menudo, un único bus de datos cumple con la función de transportar los datos en una u otra dirección, aunque nunca en ambas direcciones en forma simultánea.

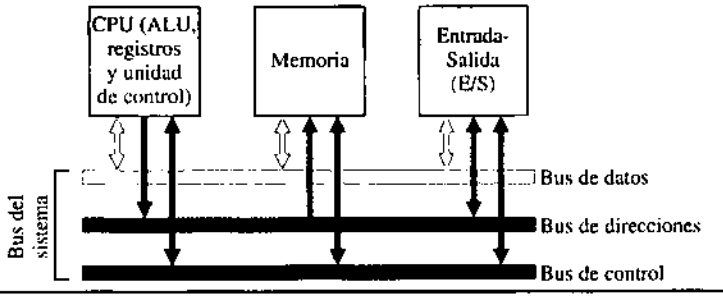


Figura 1.3 • El modelo de un sistema de computación basado en un bus. (Aportado por Donald Chiarulli, Universidad de Pittsburg.)

Si el bus tiene ser compartido por diferentes elementos que se comunican entre sí, los elementos deben tener identidades distintivas: direcciones. En algunas computadoras, todas las direcciones se suponen direcciones de memoria sin importar que formen parte de la memoria del sistema o que, en realidad, sean dispositivos de entrada-salida, mientras que en otras máquinas los dispositivos de entrada-salida tienen direcciones de entrada-salida separadas. (El tema del direccionamiento de entrada-salida se trata con más detalle en el capítulo 8, "Entradas y salidas".)

La locación o **dirección de memoria** identifica una celda de memoria en la que se almacena información, tal como se utiliza la dirección postal para identificar el lugar en que un individuo recibe o envía correspondencia. Durante una operación de lectura o escritura de memoria, el bus de direcciones contiene la dirección de la celda de memoria en la que debe leerse o escribirse el dato. Debe notarse que las expresiones "lectura" y "escritura" se plantean con respecto a la CPU: la CPU lee datos desde la memoria y los escribe en la memoria. Si se requiere leer un dato desde la memoria, el bus de datos contendrá el valor leído desde la celda de memoria seleccionada. Si la información se escribiera en memoria, el bus de datos contendría el valor del dato que se pretende almacenar en la memoria.

El bus de control es algo bastante más complejo, por lo que el análisis de este bus quedará diferido para capítulos posteriores. Por el momento, el bus de control puede considerarse como el elemento que permite la coordinación del acceso a los buses de datos y de direcciones, y la orientación de datos hacia componentes específicos.

1.5 Niveles de máquina

Tal como sucede con cualquier sistema complejo, la computadora puede verse desde diferentes perspectivas, o niveles, que van desde el nivel superior correspondiente al usuario, hasta el nivel inferior, el de los transistores. Cada uno de estos distintos niveles representa una abstracción de la computadora. Probablemente, una de las razones del enorme

éxito de la computadora digital es el alto grado de independencia o separación existente entre los niveles de abstracción. Este planteo puede apreciarse fácilmente: un usuario que necesita utilizar un programa procesador de texto requiere conocer muy poco o nada acerca de la programación de la computadora que está utilizando. En forma similar, un programador no necesita preocuparse por la estructura de los circuitos lógicos que constituyen la computadora. La separación de estos niveles ha sido explotada en forma muy interesante en el desarrollo de computadoras compatibles "hacia arriba".

1.5.1 Compatibilidad "hacia arriba"

La invención del transistor trajo como consecuencia un rápido desarrollo de la electrónica de computadoras, y juntamente con este desarrollo surgió el problema de la compatibilidad. Los usuarios de computadoras pretendían hacer uso y aprovechar las ventajas de las máquinas más modernas y más rápidas, pero cada modelo nuevo de computadora presentaba una arquitectura diferente, por lo que los viejos programas no podían funcionar sobre los nuevos circuitos. El problema de la compatibilidad entre hardware y software se agudizó tanto que los usuarios muy a menudo dilataban la compra de máquinas nuevas debido al costo que significaba reescribir los programas que debían funcionar sobre estos equipos. Cuando el usuario por fin adquiría una computadora nueva, esta podía permanecer instalada e inútilmente inactiva por meses, mientras se procedía a la reconversión de los conjuntos de datos y programas para adecuarlos a los nuevos sistemas.

En una apuesta exitosa que enfrentó a la compatibilidad con el rendimiento, IBM fue pionera en el concepto de la "familia de máquinas", aplicado en su serie 360. Las computadoras más poderosas de la misma familia podían ejecutar los programas escritos para las máquinas menos potentes, sin modificaciones en dichos programas (compatibilidad "hacia arriba"). Este tipo de compatibilidad permitió que el usuario pudiera avanzar hacia una máquina más rápida y con mayores prestaciones sin tener que reescribir los programas que funcionaban en las máquinas menos poderosas.

1.5.2 Los niveles

La figura 1.4 muestra siete niveles dentro de la computadora, comenzando desde el nivel del usuario y descendiendo hasta el nivel de los transistores. A medida que se desciende desde el nivel superior, estos niveles se tornan menos "abstractos" y comienza a aparecer, cada vez más, la estructura interna de la computadora. En los próximos puntos analizaremos estos niveles.

El nivel del usuario o del programa de aplicación

Este es el nivel de la computadora con el que nos encontramos más familiarizados. En este nivel, el usuario interactúa con la computadora por medio de la ejecución de programas tales como procesadores de texto, planillas de cálculo o juegos. Aquí, el usuario ve

a la computadora a través de los programas que en ella se ejecutan, y es poco o nada visible la estructura interna correspondiente a su nivel inferior.

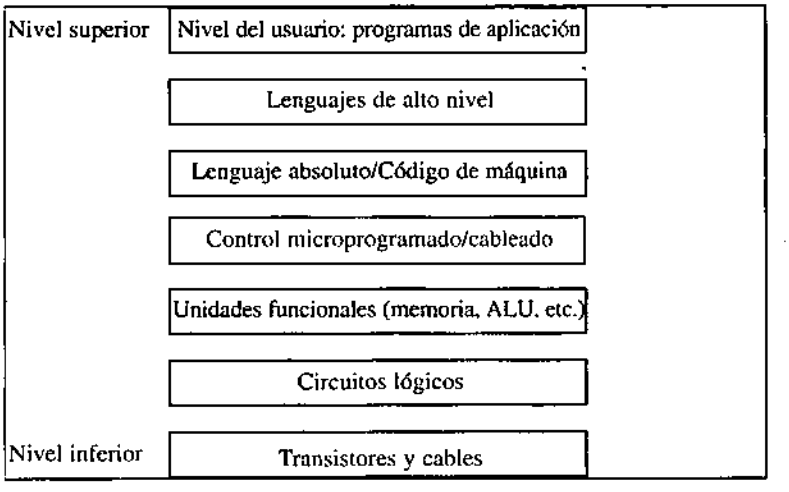


Figura 1.4 • Niveles de máquina en la jerarquía de un sistema.

El nivel del lenguaje de alto nivel

Cualquiera que haya programado una computadora en lenguajes de alto nivel, como C, Pascal, Fortran o Java, ha interactuado con la computadora en este nivel. Aquí, el programador solo ve el lenguaje y no ve ninguno de los detalles de los niveles inferiores de la máquina. En este nivel, el programador ve los tipos de datos y las instrucciones del lenguaje de alto nivel pero no necesita conocimiento alguno acerca de la forma en que la máquina configura realmente estos tipos de datos. Es función del **compilador** el convertir los tipos de datos y las instrucciones desde el lenguaje de alto nivel hacia los circuitos de la computadora. Los programas escritos en lenguaje de alto nivel pueden ser recompilados para distintos tipos de máquina, las que (al menos es lo que se espera) funcionarán de la misma manera y entregarán los mismos resultados, independientemente de la máquina para la cual fueron compilados y en la cual se ejecutaron. Puede decirse que un programa es compatible con distintos tipos de máquina si está escrito en un lenguaje de alto nivel; esta clase de compatibilidad se conoce como **compatibilidad de código fuente**.

El nivel del lenguaje de máquina

Tal como se ha mencionado previamente, el nivel de los lenguajes de alto nivel tiene muy poco que ver con la máquina en la cual dicho lenguaje de alto nivel está siendo traducido. El compilador convierte el código fuente en las verdaderas instrucciones de la máquina, a las que se suele denominar **lenguaje de máquina** o **código de máquina**. Los lenguajes de alto nivel sirven al programador dado que le proveen un conjunto cierto de tipos de datos y

construcciones de lenguaje, supuestamente bien pensado. Los lenguajes de máquina miran "hacia abajo" en la jerarquía, y así alimentan las necesidades de los aspectos de bajo nivel del diseño de la máquina. Como resultado, los lenguajes de máquina deben tratar con cuestiones circuitales tales como la estructura de los registros y la trasferencia de datos entre los registros. De hecho, muchas instrucciones pueden describirse en términos de las trasferencias efectuadas entre registros. El conjunto de instrucciones del lenguaje de máquina para una computadora dada suele denominarse **juego de instrucciones** de esa máquina.

Por supuesto, el código real que utiliza la máquina no es más que un conjunto de ceros y unos, a los que suele denominarse **código binario** o lenguaje de máquina. Como puede imaginarse, la programación en ceros y unos es un proceso tedioso y factible de generar errores. Como consecuencia, uno de los primeros programas escritos para computadora fue el llamado ***assembler***, que traduce esquemas nemotécnicos de un lenguaje común, tales como `MOVE Data, Acc`, a sus correspondientes expresiones formadas por ceros y unos en el lenguaje de máquina. Este lenguaje, cuyas construcciones presentan una relación uno a uno con el lenguaje de máquina, se conoce como **lenguaje ensamblador** (*assembler*).

Como resultado de la separación de niveles, es posible tener máquinas que difieren en la implementación del nivel inferior pero que presentan el mismo conjunto de instrucciones o algún subconjunto o superconjunto de un juego de instrucciones dado. Este planteo permitió que IBM diseñara una línea de productos como la serie **IBM 360**, que garantiza la compatibilidad hacia arriba entre los códigos de máquina. Un código de máquina que funcione sobre una 360 Modelo 35 podrá funcionar, sin modificaciones, en una IBM Modelo 50, si el usuario requiriese actualizar su equipamiento hacia la máquina más poderosa. Este tipo de compatibilidad se conoce como "compatibilidad binaria", dado que el código binario funciona, sin modificaciones, en los distintos miembros de la familia. Esta particularidad fue, en gran parte, responsable del enorme éxito de la serie de computadoras IBM 360.

Intel Corporation es otro ejemplo de compatibilidad binaria entre los integrantes de sus familias. En este caso, los códigos binarios escritos para el miembro inicial de una familia, como el 8086, funcionarán intactos en todos los integrantes posteriores, como el 80186, el 80286, el 80386, el 80486, y también en el más moderno de la familia, el procesador Pentium. Por supuesto, este enfoque no toma en cuenta la existencia de otras computadoras que ofrecen a sus usuarios distintos juegos de instrucciones, lo que hace difícil el transporte del paquete de programas ya instalado en una familia de computadoras hacia otra.

El nivel del control

Las trasferencias entre registros –mencionadas en párrafos anteriores– se llevan a cabo desde la **unidad de control**, a través de **señales de control** que transfieren la información entre un registro y otro, incluyendo, probablemente, alguna lógica circuital que pueda realizar algún tipo de transformación de la misma. La unidad de control interpreta las instrucciones de máquina una a una, lo que provoca la ocurrencia de las trasferencias entre registros o de alguna otra acción.

La forma en que esto se lleva a cabo no debe ser motivo de preocupación para el programador en lenguaje ensamblador. Los distintos integrantes de la familia de procesadores Intel 80X86 presentan la misma conducta ante un programador en lenguaje ensamblador, independientemente de cuál sea el procesador analizado. Esto se debe a que cada integrante que se incorpora a la familia se diseña para que sea capaz de ejecutar las instrucciones originales del 8086, además de cualquier instrucción nueva que hubiese sido implementada para ese integrante de la familia en particular.

Como se indica en la figura 1.4, hay diferentes formas de implementar la unidad de control. Probablemente, la forma más popular en la actualidad sea la de "cablear" la unidad de control. Esto significa que las señales de control que efectúan las transferencias entre registros están generadas a partir de un bloque de componentes lógicos digitales. Las unidades de control cableadas tienen como ventajas la velocidad y la cantidad de componentes, pero hasta no hace mucho eran extremadamente difíciles de diseñar y de modificar. (Estas técnicas se analizarán en detalle en el capítulo 6.)

Una aproximación algo más lenta pero más sencilla consiste en implementar las instrucciones como **microprograma**. Un microprograma es, en realidad, un pequeño programa escrito en un lenguaje de nivel menor aún, e implementado en los circuitos de la máquina, cuya función es la de interpretar las instrucciones del lenguaje de maquina. Este microprograma suele conocerse como ***firmware*** debido a que incluye tanto hardware como software. El *firmware* se ejecuta a través de un **microcontrolador**, el cual ejecuta las microinstrucciones reales. (La técnica de microprogramación también se explora en el capítulo 6.)

El nivel de las unidades funcionales

Las transferencias de registros y las demás operaciones implementadas por la unidad de control mueven información desde y hacia "unidades funcionales", así llamadas debido a que realizan cierta función importante para la operación de la computadora. Entre las unidades funcionales pueden incluirse los registros internos de la CPU, la ALU y la memoria principal de la computadora.

Circuitos lógicos, transistores y cables

Los niveles más bajos en los que puede observarse algún resabio del funcionamiento de los niveles superiores de la computadora corresponden a los niveles de los **circuitos lógicos** y de los **transistores**. Esto es así dado que son los circuitos lógicos los que se utilizan para construir las unidades funcionales, y porque son los transistores los que se usan para construir los circuitos lógicos. Los circuitos lógicos implementan las operaciones lógicas de más bajo nivel, de las cuales depende el funcionamiento de la computadora. En el último de los niveles, una computadora está formada por componentes eléctricos tales como transistores y cables, los que constituyen dichos circuitos lógicos; pero, a este nivel, el funcionamiento de la computadora se dispersa en términos de tensiones, corrientes, tiempos de propagación de las señales, efectos cuánticos y otros temas de bajo nivel.

Interacciones entre niveles

Las distinciones dentro de un nivel dado y entre niveles suelen confundirse en forma muy frecuente. Por ejemplo, una arquitectura nueva de computadora puede contener instrucciones de punto flotante en su implementación más completa, pero en una implementación mínima puede tener solo hardware suficiente como para ejecutar instrucciones con variables enteras. Las instrucciones de punto flotante se **capturan**[1] antes de la ejecución y se reemplazan por una secuencia de instrucciones de lenguaje de máquina que imitan o **emulan** esas instrucciones de punto flotante utilizando las instrucciones de formato entero existentes. Este es el caso de aquellos **microprocesadores** que utilizan coprocesadores de punto flotante opcionales. Los que no cuentan con el coprocesador de punto flotante emulan las instrucciones de punto flotante por medio de una serie de rutinas de punto flotante que se encuentran implementadas en el lenguaje de máquina del microprocesador y almacenadas en un circuito integrado de memoria **ROM**, que identifica a una memoria solo de lectura. Los puntos de vista de los lenguajes absoluto y de alto nivel son los mismos para ambas implementaciones, con excepción de la velocidad de ejecución.

Es posible llevar esta emulación al extremo de emular el juego completo de instrucciones de una computadora sobre otra computadora distinta. Los programas que cumplen con esta función se conocen como **emuladores** y fueron utilizados por **Apple Computer** para mantener la compatibilidad del código binario cuando Apple comenzó a emplear los circuitos integrados PowerPC de Motorola en lugar de los circuitos integrados 68000, cuyo juego de instrucciones es completamente distinto.

El nivel del lenguaje de alto nivel y los niveles del *firmware* y de las unidades funcionales pueden estar tan entrelazados que resulte difícil identificar qué operación se está produciendo en qué nivel. El valor de la división de la arquitectura de computadoras en un conjunto de niveles jerárquicos no es tanto el de la clasificación, que según acabamos de ver a veces puede ser dificultosa, sino, simplemente, el de permitir ciertos enfoques en el análisis a llevar a cabo en los capítulos que siguen.

La perspectiva del programador. La arquitectura del juego de instrucciones

Según se ha descrito en el análisis de niveles previo, el programador que programa en lenguaje de máquina tiene interés en el lenguaje y en las unidades funcionales de la máquina. Este conjunto de elementos, formado por el juego de instrucciones y las unidades funcionales, se conoce como **arquitectura de programación** (ISA, *instruction set architecture*) de la computadora.

1. El concepto se analiza en el capítulo 6.

La perspectiva del arquitecto de computadoras

El arquitecto de computadoras, por otra parte, observa al sistema desde todos sus niveles. El arquitecto que enfoca el diseño de una computadora está sujeto invariablemente a requerimientos de rendimiento y restricciones de costos. El rendimiento puede estar especificado a partir de la velocidad de ejecución de los programas, de la capacidad de almacenamiento de la máquina o de una variedad de parámetros diferentes. El costo puede estar planteado en términos monetarios, o de peso y tamaño, o de consumo de potencia. El diseño propuesto por el arquitecto de computadoras debe intentar cumplir con las perspectivas especificadas para el rendimiento, pero sin salirse de los límites de costos establecidos, lo que normalmente lleva a soluciones de compromiso entre y a través de los niveles de la máquina.

1.6 Un sistema de computación típico

Las computadoras modernas han progresado desde los grandes monstruos de las décadas de 1950 y 1960 hasta llegar a las computadoras, mucho más pequeñas y más poderosas, que nos rodean en la actualidad. Aun con todos los grandes avances de la tecnología de computadoras ocurridos en las ultimas décadas, las cinco unidades básicas del modelo von Neumann siguen siendo visibles en las computadoras modernas.

La figura 1.5 muestra la configuración típica de una computadora de escritorio, cuya unidad de entrada consiste en un **teclado**, a través del cual el usuario ingresa datos y comandos. Un **monitor de video** constituye la unidad de salida, en la que se presenta la información en forma visual. La unidad de control y la ALU se agrupan dentro de un microprocesador único que funciona como CPU del sistema. La unidad de memoria consta de un conjunto de circuitos individuales de memoria, así como de unidades de **disco rígido**, de **disquete** y de **CD-ROM**.

Una observación del interior de la máquina permite ver que su corazón se encuentra contenido en una única placa de circuito impreso, conocida comúnmente como **placa madre** (*motherboard*), similar a la que se muestra en la figura 1.6. La placa madre contiene **circuitos integrados**, conectores para insertar placas de expansión, y cables que interconectan los circuitos integrados y los conectores de expansión. En la figura se destacan las secciones de entrada, salida, memoria, control y unidad aritmético-lógica. (En capítulos posteriores se desarrollan los temas asociados.)

1.7 Organización de la obra

El funcionamiento interno de las computadoras se desarrolla en los capítulos siguientes. El capítulo 2 analiza la representación de los datos, lo que provee el fundamento para todos los capítulos que siguen. El capítulo 3 analiza los métodos relacionados con la

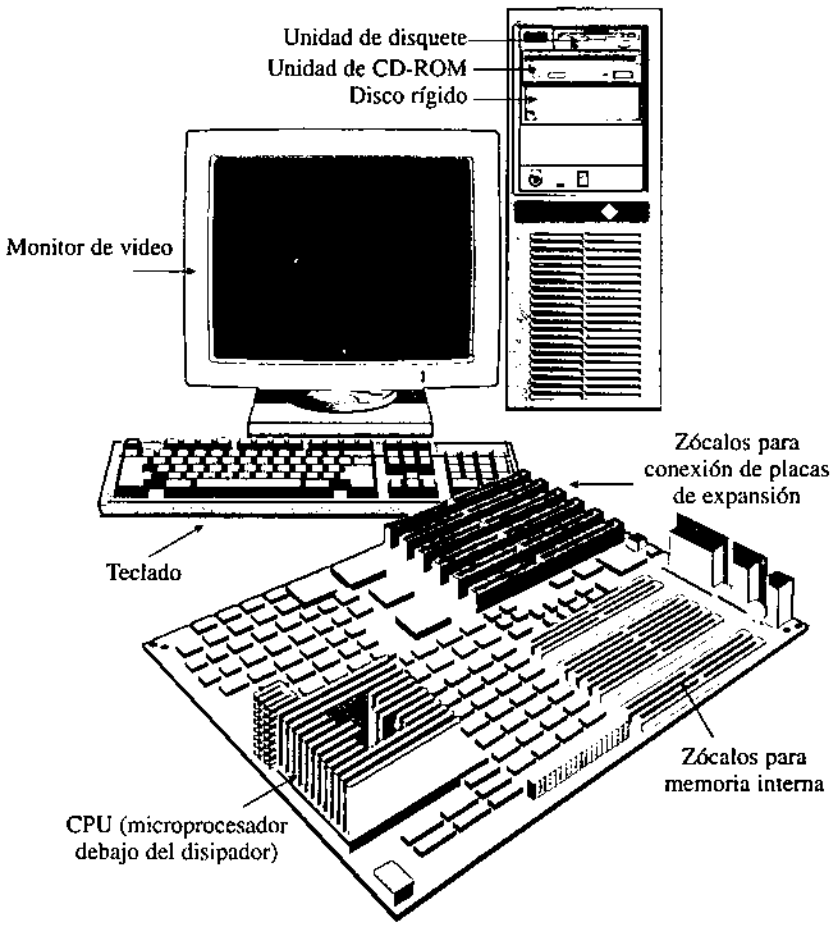


Figura 1.5 • Un sistema de computación de escritorio.

implementación del cálculo aritmético. Los capítulos 4 y 5 analizan la arquitectura de programación, lo que servirá como base para entender la interacción entre los componentes de un sistema de computación. El capítulo 6 vincula los contenidos anteriores para encarar el diseño y el análisis de una unidad de control para la arquitectura del juego de instrucciones. En el capítulo 7 se cubre la organización de las unidades de memoria, así como las técnicas de administración de la memoria. En el capítulo 8, se tratan los temas referidos a entrada, salida y comunicaciones. El capítulo 9 analiza aspectos avanzados de los sistemas de CPU única (que podrían tener más de una unidad de procesamiento). En el capítulo 10 se consideran aspectos referidos a sistemas de múltiples CPU, tales como las llamadas arquitecturas paralelo y distribuida y las arquitecturas de redes. Por último, en los apéndices A y B se echa una mirada al diseño de circuitos lógicos digitales, bloques fundamentales para los componentes básicos de la computadora.

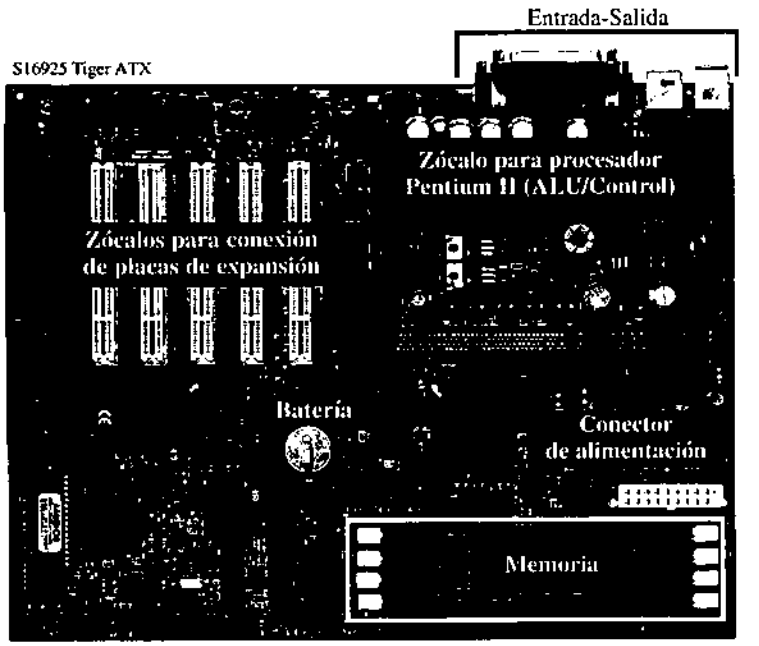


Figura 1.6 • Una placa madre para procesador Pentium II. (**Fuente:** TYAN Computer, http://tyan.com/.)

1.8 Estudio de un caso: ¿Qué le ocurrió a las supercomputadoras?

[(*Nota de los autores:* El aporte que sigue proviene de la página http://www.paralogos.com/DeadSuper, creada por Kevin D. Kissell, kevink@acm.org. El sitio web creado por Kissell detalla docenas de proyectos de supercomputadoras que fueron quedando por el camino. (Véase la figura 1.7[2] para tener una visión histórica de las primeras tecnologías de supercomputadoras.) Una de las razones principales que han llevado a las supercomputadoras a su casi extinción es el hecho de que las computadoras comunes de hoy en día logran una parte significativa de la potencia de las supercomputadoras a un precio accesible para el individuo común. La relación entre el precio de una computadora de escritorio y su rendimiento es muy favorable debido a los bajos costos logrados a través de las ventas en el mercado masivo. Dado que las supercomputadoras no tienen acceso a ese mercado masivo, mantienen una relación muy alta entre el precio y el rendimiento.

El trabajo que sigue, escrito por Kissell y extraído de un artículo publicado en *Electrical Engineering Times*, destaca las enormes inversiones que se realizan en el desarrollo de los microprocesadores de uso diario, lo que ayuda a mantener la mencionada relación favorable entre precio y rendimiento en las computadoras de escritorio de bajo costo.]

2. Figura incluida por cortesía de la Universidad de Manchester. La historia de Manchester Mark I se detalla en http://www.computer50.org.

Figura 1.7 • La computadora Manchester University Mark I, puesta en operación el 21 de junio de 1948. (No debe confundirse con Harvard Mark 1, donada a la Universidad de Harvard por IBM, en agosto de 1944.) (Reproducido con autorización del Departamento de Ciencias de la Computación de la Universidad de Manchester.)

¿El final de una época dorada?

Desde la construcción de las primeras computadoras programadas hasta mediados de la década de 1990, siempre hubo lugar en la industria de la computación para cualquiera que tuviera alguna idea brillante, aun cuando fuese desafiante, acerca de cómo desarrollar una máquina más poderosa. La computación se convirtió en estratégica durante la Segunda Guerra Mundial, y se mantuvo en tal situación durante la época de la Guerra Fría que siguió a aquella. La computación de alto rendimiento es esencial para cualquier programa moderno de armamento nuclear, por lo que la carrera de la tecnología de computadoras fue una consecuencia lógica de la carrera armamentista. Si bien las computadoras poderosas son de mucho valor en una gran cantidad de sectores industriales, como las industrias del petróleo, química, médica, aeronáutica, automotriz y de ingeniería civil, será muy difícil desmentir el papel de los gobiernos, y en particular el de los laboratorios nacionales del gobierno estadounidense, como catalizadores e incubadores de las innovaciones en las tecnologías de computadoras. La industria privada puede llegar a comprar mayor número de máquinas pero muy rara vez asumirá el riesgo de comprar una máquina cuyo número de serie sea de un solo dígito. La desaparición del comunismo soviético y el final de la Guerra Fría provocaron, como consecuencia, un mundo en general más seguro y más próspero, pero eliminaron la razón de ser de muchos mercaderes del rendimiento a cualquier precio.

Junto con estos cambios geopolíticos aparecieron algunos hechos tecnológicos y económicos que representaron problemas para los productores especializados en computadoras de alta tecnología. La aparición de los microprocesadores en la década de 1970 planteó como punto principal la posibilidad de colocar una computadora de programa almacenado en un único pedazo de silicio. Las presiones de la competencia, y el deseo de generar ventas mediante la obsolescencia del producto del año anterior, dieron como resultado la duplicación de la potencia de computación de los microprocesadores cada 18 meses, según el enunciado de la conocida "ley de Moore". En su camino, los di-

señadores de microprocesadores tomaron prestadas casi todas las argucias y técnicas utilizadas en el pasado por los diseñadores de supercomputadoras y *mainframes*: niveles jerárquicos de almacenamiento, segmentaciones, unidades funcionales múltiples, multiprocesamiento, ejecución desordenada, predicción de saltos, procesamiento SIMD (*Single Instruction, Multiple Data*), ejecución especulativa. A mediados de los años noventa, las ideas de los investigadores pasaban directamente de la etapa de simulación a la implementación de los microprocesadores destinados a las computadoras de uso masivo. No obstante, debe notarse que la mayor parte de los avances logrados en el rendimiento de los microprocesadores, en la década pasada, no llegó desde estas técnicas sofisticadas de la arquitectura de computadoras, sino, simplemente, desde el aumento de las velocidades del reloj de los procesadores y de los crecimientos cuantitativos en los recursos de los procesadores derivados de los avances de la tecnología semiconductora. Alrededor de 1998, la CPU de la mejor computadora personal basada en Windows funcionaba ya a una frecuencia de reloj mayor que la de la supercomputadora más importante de la línea Cray Research en 1994.

No puede sorprender entonces que las políticas de los laboratorios nacionales de los Estados Unidos se hayan desplazado desde la adquisición de sistemas desarrollados desde su base para tomar la forma de supercomputadoras hacia el despliegue de grandes ensambles de sistemas producidos en masa y basados en microprocesadores, con el proyecto ASCI como emblema de esta actividad. A la fecha de este trabajo, queda por ver si tales aglomeraciones resultarán ser lo suficientemente estables y apropiadas para el trabajo de producción, pero los resultados preliminares han sido, al menos, satisfactorios. Los días felices de las supercomputadoras basadas en tecnologías exóticas y en arquitecturas innovadoras bien pueden haber llegado a su fin.

Inversión o muerte: La vida de Intel en la orilla[3] **por Roy Wilson y Brian Fuller**

Santa Clara, California – Con alrededor de seiscientos millones de dólares a ser utilizados como inversión durante este año, Intel Corp. se ha unido a las grandes ligas de las empresas de capital empresario. Pero el único imperativo que lleva al gigante de los microprocesadores a invertir le da una influencia desproporcionada aun con respecto a esa enorme cifra. Para Intel, las inversiones de capital no son solo una fuente de ingresos; son una herramienta vital en la lucha por sobrevivir.

La supervivencia parecería ser una preocupación poco importante para la empresa de semiconductores más grande del mundo. Pero Intel, en un camino que le es totalmente propio, vive colgado en el equilibrio. Para cada nueva generación de procesadores, Intel debe hacer enormes inversiones en el desarrollo de procesos, en edificios y en fábricas, una inversión demasiado gigante como para perderla.

3. Derechos reservados ©1998 CMP Media Inc. reproducido con autorización de *Electronic Engineering Times*.

Gordon Moore, el emérito presidente de Intel, le pone la escala a la apuesta. "Una planta de investigación y desarrollo cuesta hoy cuatrocientos millones solo en el edificio. Luego, hay que ponerle alrededor de mil millones de dólares en equipamiento. Esto representa una fábrica para un cuarto de micrón que puede llegar a producir alrededor de 5.000 obleas por semana, lo que se dice la planta utilizable más pequeña. Para la próxima generación," afirma Moore, "la inversión mínima será de dos mil millones, con entre tres y cuatro mil millones necesarios para cualquier tipo de volumen de producción. Ninguna otra industria tiene una vida tan corta con inversiones tan enormes."

Gran parte de estas cifras será gastada antes de que haya una necesidad concreta de los microprocesadores que la planta vaya a producir. En esencia, el total de los 4.000 millones por planta es una apuesta basada en la idea de que la industria absorberá un gran número de CPU a buen precio solo porque sean algo más rápidos que los elementos disponibles actualmente. Si solo por una generación dicha situación no ocurriese (si alguien considerara, por ejemplo, que el Pentium II es lo suficientemente rápido, gracias), los resultados serían impensables.

"Mi pesadilla consiste en despertarme algún día y no necesitar mayor poder de cómputo," dijo Moore.

Resumen

La arquitectura de las computadoras trata aquellos aspectos de una computadora que son visibles para el programador, en tanto que la organización de las computadoras trata aquellos aspectos que se encuentran en un nivel más físico y que no aparecen como visibles para el programador. Históricamente, los programadores debían enfrentarse con cada aspecto de una computadora: Babbage con elementos mecánicos y los programadores de ENIAC con cables y paneles de conexión. A medida que crece el nivel de sofisticación de las computadoras, el concepto de niveles de máquina aparece en forma más pronunciada, permitiendo que las computadoras presenten conductas internas y externas muy diferentes a la vez que se administra la complejidad en niveles estratificados. El desarrollo individual más significativo que hace posible este planteo es el concepto de computadora de programa almacenado, que toma cuerpo en el modelo von Neumann. Es el modelo von Neumann el que se ve en la mayor parte de las computadoras corrientes hoy en día.

Para lectura posterior

La historia de la computación está llena de personalidades e hitos interesantes. Harlan Anderson ofrece un detalle corto y ameno de unos y de otros correspondientes al último siglo. La obra de Charles Bashe y otros ofrece un resumen interesante acerca de las máquinas IBM. A. Bromley, hace una crónica de las máquinas de Babbage. A. Ralston y E. Reilly aportan biografías cortas de las

personalidades más célebres. B. Randell cubre la historia de las computadoras digitales. Una historia de la computación, de lectura accesible, se encuentra en el sitio web http://www.ifi.unizh.ch/se/people/hoyle/Lecture/. Doron Swade, cubre una versión legible del método de las diferencias finitas tal como aparece en las máquinas de Babbage y en la versión de la máquina diferencial analítica creada por el Museo de Ciencias de Londres. El trabajo de A. Tanenbaum es uno de una cantidad de textos que popularizó la noción de niveles dentro de las máquinas.

Anderson, Harlan, "Dedication address for the Digital Computer Laboratory at the University of Illinois", 17 de abril de 1991, tal como se reproduce en: *IEEE Circuits and Systems Society Newsletter*, vol. 2, nº 1, marzo de 1991, p.p. 3-6.

Bashe, Charles J., Lyle R. Johnson, John H. Palmer y Emerson W. Pugh, *IBM's Early Computers*, The MIT Press, 1986.

Bromley, A. G., "The evolution of Babbage's Calculating Engines", *Annals of the History of Computing*, vol. 9, 1987, p.p. 113-138.

Randell, B., *The Origins of Digital Computers*, 3ª ed., Springer-Verlag, 1982.

Ralston, A. y E. D. Reilly, eds., *Encyclopedia of Computer Science*, 3ª ed., van Nostrand Reinhold, 1993.

Doron D. Swade, "Redeeming Charles Babbage's Mechanical Computer", en: *Scientific American*, vol. 268, nº 2, febrero de 1993, p.p. 86-91.

Tanenbaum, A., *Structured Computer Organization*, 4ª ed., Prentice Hall, 1999. (Traducción al español disponible: *Organización de computadoras*, Prentice Hall, 2000.)

Problemas

1.1 La ley de Moore, que se atribuye al creador de Intel, Gordon Moore, dice que la potencia de computación se duplica cada 18 meses por el mismo precio. Como observación no relacionada surge que las instrucciones de punto flotante se ejecutan en forma circuital cien veces más rápido que cuando se las emula. Usando la ley de Moore como guía, ¿cuánto tiempo le llevará a la potencia de computación crecer a punto tal que las instrucciones de punto flotante se emulen tan rápido como sus anteriores contrapartidas en el hardware?

Capítulo 2

Representación de la información

2.1 Introducción

En los primeros días de la computación eran frecuentes los errores de concepto referidos a las computadoras. Uno de los más habituales consistía en creer que la computadora era únicamente una máquina sumadora gigante capaz de realizar operaciones aritméticas. Las computadoras tenían capacidad para hacer mucho más que eso, aun en aquellos primeros días. Otro error de concepto, en contradicción con el anterior, era el de pensar que la computadora podía hacer "cualquier cosa". Sabemos ahora que hay ciertos problemas que resultan intratables, aun para la máquina más poderosa con arquitectura von Neumann. La percepción correcta, por supuesto, está en algún lugar intermedio entre ambos.

Hoy nos resultan familiares algunas operaciones que no son aritméticas, tales como la computación gráfica, el audio digital y aun el manejo del (ratón) *mouse* de la computadora. Independientemente del tipo de información que se esté manipulando, debe estar representada por patrones de unos y ceros. (También conocidos como códigos "sí-no".) Este planteo lleva inmediatamente a la pregunta referida a cómo debe representarse o describirse la información dentro de la máquina, o sea, cuál debe ser la **representación** o **codificación** de los datos. Las imágenes gráficas, el audio digital o las pulsaciones sobre los botones del mouse deben estar todos codificados de alguna manera sistemática y normalizada.

Dado que es la más conocida, puede pensarse en la representación decimal de la información como la forma más natural de dicha representación. No obstante, el uso de códigos sí-no en la representación de la información antecede en muchos años a la computadora, como es el caso del código Morse.

Este capítulo introduce algunos de los ejemplos de codificación más simples y más importantes: la codificación de números de punto fijo con y sin signo, la de los números reales (conocidos en la jerga de la computación como **números de punto flotante**), y la de los caracteres requeridos para la impresión de texto. Se podrá ver que en todos los casos existen diversas formas de codificar los distintos tipos de datos, algunas de ellas útiles en un contexto dado, otras, más útiles en otro contexto. Asimismo, este capítulo presentará una primera mirada sobre la aritmética de computadoras, con el objeto de entender algunos de los esquemas de codificación, si bien los detalles quedarán para ser tratados en el capítulo 3.

Un punto clave en el desarrollo de un formato de representación de información en una computadora es la determinación del espacio de almacenamiento que se deberá dedicar a cada tipo de dato. Por ejemplo, un diseñador de computadoras puede decidir representar los números enteros con 32 bits, e implementar una ALU que pueda realizar cálculo aritmético sobre esos conjuntos de 32 bits, y que devuelva resultados también en 32 bits. No obstante, algunos números pueden ser muy grandes como para ser representados en 32 bits, y en otros casos, los operandos pueden representarse por 32 bits pero el resultado de una operación ser mayor, lo que provocará una condición de **desborde***, condición cuyo significado se describe en el capítulo 3. Por consiguiente, se hace necesario tener en claro cuáles son los límites a imponer sobre la precisión y el rango de las operaciones aritméticas derivados de la naturaleza finita de las representaciones utilizadas. Estas limitaciones se analizan en las secciones siguientes.

2.2 Números de punto fijo

En la representación de números de punto fijo**, todos los números a representar tienen exactamente la misma cantidad de dígitos y la coma decimal está siempre ubicada en el mismo lugar. Como ejemplos dentro del sistema decimal pueden plantearse los números "0,23", "5,12" y "9,11". En estos ejemplos, cada número tiene tres dígitos y la coma decimal se ubica a continuación del primero de ellos. Los ejemplos que pueden presentarse en el sistema **binario** de numeración (en el que cada dígito solo puede adoptar los valores 0 o 1) son "11,10", "01,10" y "00,11", en los que hay cuatro dígitos y donde la coma "decimal" está entre el segundo y el tercero de ellos. La diferencia principal entre la representación de números de punto fijo en el papel y la forma en que se almacenan dentro de la computadora es que, al representar números en este formato, **no se almacena coma decimal alguna,** sino que, simplemente, se supone que ocupa un lugar determinado. Podría afirmarse que la coma decimal solo existe en la mente del programador.

El estudio de la representación de números en el formato de punto fijo se inicia con el análisis del rango y la precisión de esta representación, utilizando para esto el sistema de numeración decimal. Continúa con una mirada a la naturaleza de los sistemas de numeración, tales como el decimal y el binario, y a los métodos de conversión entre diferentes sistemas. Con estos fundamentos, se investigan distintas formas de representación de números negativos expresados en formato de punto fijo, analizando, finalmente, algunas operaciones matemáticas simples que pueden realizarse con ellos.

---

* *N. de T.*: En muchos países de habla hispana, suele utilizarse el término original del idioma inglés (***overflow***) sin traducción alguna.

** *N. de T.*: Se aceptará en este texto la nomenclatura habitual en la que se identifica como "punto" a la coma decimal, debiendo quedar claro que las menciones al punto fijo o flotante se refieren a la ubicación de dicha coma decimal.

2.2.1 Rango y precisión en números de punto fijo

Una representación de punto fijo puede caracterizarse por el **rango** de los números que expresa (dado por la diferencia entre el número mayor y el menor) y por su **precisión** (la distancia entre dos números consecutivos en una serie numérica). Para el ejemplo planteado anteriormente, en el sistema de numeración decimal, si se utilizan tres dígitos con la coma decimal entre el primero y el segundo, el rango de representación varía desde 0,00 hasta 9,99, incluyendo los extremos, lo que se expresa como [0,00..9,99]. La precisión de la representación es 0,01, y el **error** resultante puede considerarse como la mitad de la diferencia entre dos números consecutivos, tales como 5,01 y 5,02, entre los que hay una diferencia de 0,01. El error, por ende, es 0,01/2 = 0,005. Esto significa que cualquier número del rango 0,00 a 9,99 puede representarse en este formato con una aproximación de hasta 0,005 de su valor real o preciso.

Nótese de qué manera existe un compromiso entre rango y precisión. Si la coma decimal se coloca a la derecha del número, el rango pasa a ser [000..999] pero la precisión pasa a ser de 1,0. Si, en cambio, la coma decimal se coloca a la izquierda del número representado, el rango pasa a ser [0,000..0,999] y la precisión pasa a valer 0,001.

En cualquiera de los casos, hay solo 10^3 "objetos" decimales diferentes, cuyo rango va desde 000 a 999 o desde 0,000 hasta 0,999, en consecuencia, solo es posible representar 1000 elementos diferentes, independientemente de cómo se administren rango y precisión.

No hay razón alguna para que el rango empiece en 0. Un número decimal de dos dígitos puede tener rangos de [00..99] o de [-50..+49], o aun de [-99..0]. El apartado 2.2.6 analiza con mayor profundidad la representación de los números negativos.

El rango y la precisión son conceptos importantes en la arquitectura de computadoras, debido a que ambos son elementos finitos en la implementación de la arquitectura, en tanto que son infinitos en el mundo real, por lo que el usuario debe tener en claro las limitaciones que surgen al tratar de representar información externa en el formato interno.

2.2.2 La ley asociativa del álgebra no siempre funciona en la computadora

Entre los primeros conceptos que se plantean en matemática, aparece la llamada propiedad o ley asociativa:

$$A + (B + C) = (A + B) + C$$

Como podrá verse, la ley asociativa del álgebra no funciona cuando se representan números en el formato de punto fijo con representación finita. Considérese una representación de números enteros en formato de punto fijo con un solo dígito decimal y la coma decimal a la derecha, con un rango de [-9..9]. En el caso en que $A = 7$, $B = 4$ y $C = -3$, la suma de

$A + (B + C) = 7 + (4 - 3) = 7 + 1 = 8$. Pero $(A + B) + C = (7 + 4) - 3 = 11 - 3$. El resultado intermedio, 11, ¡está fuera del rango de la representación considerada! Si bien el resultado final podría haber quedado dentro del rango de representación del sistema de numeración, se ha producido un desborde en el resultado intermedio. Este ejemplo permite ver que, si algún resultado intermedio resulta incorrecto, el resultado final de la operación completa también es incorrecto.

Por consiguiente, surge como conclusión que la ley asociativa del álgebra no se cumple cuando se representa la información numérica en formato de punto fijo de longitud finita. Se trata de una consecuencia inevitable de esta forma de representación y no hay solución práctica excepto la de detectar el desborde en el momento en el que ocurra, o, habiendo detectado el desborde, repetir la operación con una representación que permita un rango mayor. (Esta última técnica se usa muy raramente, excepto en situaciones críticas.)

2.2.3 Sistemas de numeración posicionales

Esta sección analiza de la utilización de sistemas numéricos con bases arbitrarias, si bien el enfoque se concentra sobre los sistemas más utilizados en las computadoras digitales, tales como el sistema binario (de base 2) y sus parientes cercanos, los sistemas octal (de base 8) y hexadecimal (de base 16).

La **base o raíz** de un sistema de numeración define el rango de valores posibles que pueden adoptar sus dígitos. En el sistema de numeración decimal (de base 10), los diferentes dígitos de una cantidad numérica se representan por alguno de los 10 valores correspondientes al 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. El sistema naturalmente utilizado para la representación de números en una computadora es el sistema binario, en el que se representa la información como un conjunto de ceros y unos.

La expresión general que permite determinar el valor decimal de un número en un sistema de numeración de base k y en formato de punto fijo es la siguiente:

$$\text{Valor} = \sum_{i=-m}^{n-1} b_i \cdot k^i$$

El valor del dígito que ocupa la posición i está representado por b_i. Existen en este caso n dígitos a la izquierda de la coma fraccionaria y m dígitos a su derecha. Esta forma de representación de un número, en la que cada posición tiene asignado un determinado valor, se denomina **sistema de numeración posicional**. Considérese la expresión (541,25), en la cual el subíndice 10 representa la base en la que se expresa el número.* En este caso, $n = 3$, $m = 2$ y $k = 10$:

* *N. de T.*: En el caso de números expresados en el sistema decimal, por convención, suele omitirse el subíndice indicador de la base, sobreentendiéndose que todo número así expresado lo está en dicho sistema decimal.

$$5 \times 10^2 + 4 \times 10^1 + 1 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2} =$$

$$= (500) + (40) + (1) + (2/10) + (5/100) = (541{,}25)$$

Si en forma similar se considera el número binario $(1010{,}01)_2$, en el que $n = 4$, $m = 2$ y $k = 2$, se tiene

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

$$= (8) + (0) + (2) + (0) + (0/2) + (1/4) = (10{,}25)$$

Este último procedimiento da una idea acerca de cómo convertir un número expresado en un sistema de numeración de base cualquiera al sistema de numeración decimal, mediante la utilización de una **representación polinómica**. Se trata de multiplicar cada dígito por el peso asignado a su posición (potencias de dos en este ejemplo) y luego sumar los valores para obtener el número convertido. Si bien este método es válido para conversiones de todo tipo de sistema, algunos sistemas plantean ciertas dificultades especiales, las que serán desarrolladas en la próxima sección.

Nótese que en estos sistemas de numeración posicionales se define el bit con el mayor peso asociado como **bit más significativo** (MSB, *most significant bit*), y el bit de menor peso como **bit menos significativo** (LSB, *least significant bit*). Por conversión, el LSB es el bit a la derecha de la expresión numérica, en tanto que el MSB es el bit a la izquierda de esta.

2.2.4 Conversión entre sistemas

En la sección anterior se ha planteado un ejemplo de cómo puede convertirse un número expresado en el sistema de numeración binario al sistema de numeración decimal. La conversión de un número en el sentido contrario puede ser un poco más compleja. La forma más sencilla de convertir números de punto fijo que contengan tanto parte entera como fraccionaria consiste en operar con cada una de sus partes por separado. Como ejemplo, se planteará la conversión del número 23,375 al sistema binario. Se comienza por separar el número en sus partes entera y fraccionaria:

$$23{,}375 = 23 + 0{,}375$$

Conversión de la parte entera de un número de punto fijo. Método de los restos

Según lo sugerido en la sección anterior, la forma polinómica general para la representación de un número entero binario es:

$$b_i \times 2^i + b_{i-1} \times 2^{i-1} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$$

Si se divide al número entero por dos, se obtendrá:

$$b_i \times 2^{i-1} + b_{i-1} \times 2^{i-2} + \ldots + b_1 \times 2^0$$

con un resto de b_0. Como resultado de dividir por dos el número entero original se obtiene el valor del primer coeficiente binario b_0. Puede repetirse este proceso aplicándolo al polinomio remanente para obtener el segundo coeficiente b_1. Si se continúa aplicando el mismo procedimiento en forma iterativa, se obtendrán todos los coeficientes b_i. Este procedimiento forma la base del **método de los restos** para convertir números enteros entre diferentes sistemas de numeración.

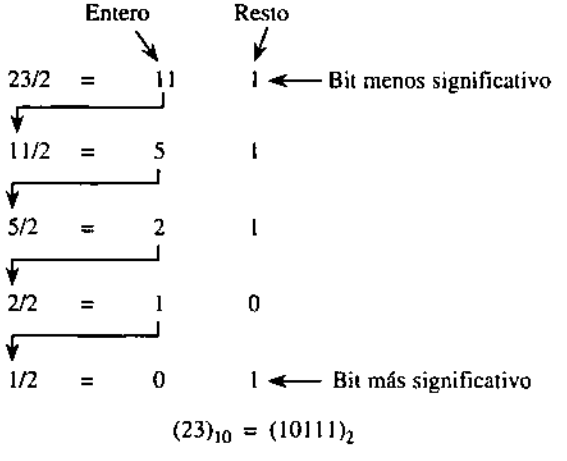


Figura 2.1 • Conversión de un número entero decimal al sistema de numeración binario mediante el método de los restos.

El método de los restos se aplicará ahora para convertir el número 23 al sistema de base 2. Como se ve en la figura 2.1, el entero se divide inicialmente por 2, lo que deja un resto de 0 o 1. En este caso, 23/2 produce un cociente de 11 y un resto de 1. Este primer resto es el dígito (bit) menos significativo del número convertido (el bit a la derecha del número). En el paso siguiente, se divide ahora el número 11 por 2, lo que deja un cociente de 5 y un resto de 1. Al dividir 5/2 se obtendrá un cociente de 2 y un resto de 1. El proceso continúa hasta obtener 0 como cociente. Si se sigue dividiendo luego de obtener cociente nulo, solo se lograrán ceros como futuros cocientes y restos, lo que no cambiará el valor del número convertido. Los restos así obtenidos se recogen en el orden indicado en la figura 2.1, para obtener la expresión binaria del número convertido, lo que en este caso da por resultado $23 = (10111)_2$. En general, todo número expresado en el sistema decimal puede convertirse a cualquier otro sistema simplemente dividiendo el entero decimal reiteradamente por la base del sistema de numeración al que se lo quiere convertir.

El resultado puede verificarse mediante la conversión del mayor número binario así obtenido al sistema decimal, mediante el método polinómico:

$$(10111)_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 =$$

$$= 16 + 0 + 4 + 2 + 1 =$$

$$= (23)$$

A esta altura, se ha procedido a la conversión de la parte entera del número 23,375 al sistema de numeración binario.

Conversión de la parte fraccionaria de un número de punto fijo. Método de las multiplicaciones

La conversión de la parte fraccionaria puede resolverse multiplicando sucesivamente la fracción por dos, de acuerdo con lo que se describe a continuación.
Una fracción binaria se representa, en su forma general, según la expresión siguiente:

$$b_{-1} \times 2^{-1} + b_{-2} \times 2^{-2} + b_{-3} \times 2^{-3} + \ldots$$

Si se multiplica dicha expresión por 2, se obtiene:

$$b_{-1} + b_{-2} \times 2^{-1} + b_{-3} \times 2^{-2} + \ldots$$

Se puede determinar así el coeficiente b_{-1}. Si se reitera el procedimiento sobre la fracción remanente, se obtendrán los sucesivos b_i. Este proceso determina la forma de convertir números fraccionarios entre distintos sistemas de numeración utilizando el **método de las multiplicaciones**. En el ejemplo aquí planteado (véase la figura 2.2), la fracción inicial, 0,375, es menor que 1. Si se la multiplica por 2, el resultado obtenido será menor que 2. El dígito a la izquierda de la coma fraccionaria será entonces 0 o 1. Este es el primer dígito a la derecha de la coma fraccionaria en el número convertido a base 2, tal como se muestra en la figura. Se repite el proceso sobre la parte fraccionaria hasta que se obtenga una fracción nula, en cuyo caso las siguientes iteraciones solo darán por resultado ceros adicionales, o hasta que se haya alcanzado el límite de precisión requerido por la representación utilizada. Se recogen los dígitos obteniéndose así el resultado: $0,375 = (0,011)_2$.

En este procedimiento, el multiplicador coincide con la base del sistema numérico de destino. En este caso, el multiplicador es 2, pero si se pretendiese realizar una conversión a otro sistema, como podría ser un sistema de base 3, se utilizaría 3 como multiplicador.[1]

1. Alternativamente, puede usarse el sistema de numeración decimal y evitar la conversión mediante la representación binaria de los diez dígitos decimales. Esta representación se conoce como BCD (decimal codificado en binario) y se la describirá más adelante.

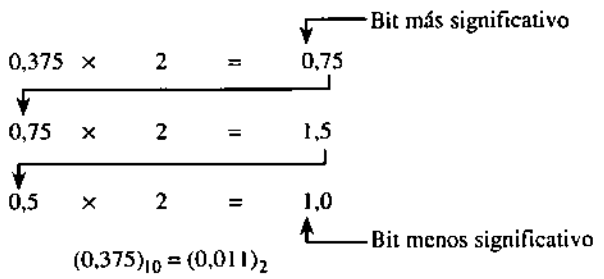


Figura 2.2 • Conversión de una fracción decimal al sistema binario por medio del método de las multiplicaciones.

Puede verificarse el resultado de la conversión si se realiza la reconversión al sistema decimal del número binario así obtenido, utilizando la representación polinómica:

$$(0,011)_2 = 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 0 + 0,25 + 0,125 = (0,375)_{10}$$

Si ahora se combinan las partes entera y fraccionaria, se obtiene el resultado final:

$$23,375 = (10111,011)_2$$

Fracciones no exactas

Si bien el método de conversión funciona con todos los sistemas de numeración, el proceso puede llegar a producir pérdidas de precisión. Por ejemplo, no todas las fracciones representadas en el sistema de numeración decimal pueden tener como equivalente un número racional exacto en el sistema binario. Considérese como ejemplo la conversión del número $0,2_{10}$ al sistema binario de numeración, siguiendo el procedimiento expresado, y de acuerdo con lo que se observa en la figura 2.3. En el cuarto paso de la conversión reaparece la fracción 0,2, por lo que a partir de ese punto el proceso se repite en forma periódica.

Para tratar de justificar la causa de esta situación, considérese que cualquier fracción exacta expresada en el sistema binario puede representarse como $i/2^k$ para ciertos valores de los enteros i y k. (Esta forma de representación no es válida para números fraccionarios periódicos.) En forma algebraica:

$$2^k = i \times 5^k/(2^k \times 5^k) = i \times 5^k/10^k = j/10^k$$

En la que j es el entero $i \times 5^k$. La fracción no es periódica en el sistema decimal. Este concepto reafirma el hecho de que solo las fracciones no periódicas de un sistema de numeración de base b pueden presentarse como i/b^k para algunos valores enteros de i y k. La condición que debe cumplir una fracción decimal exacta para que su equivalente binario también sea exacto es:

$$i/10^k = i/(5^k \times 2^k) = j/2^k$$

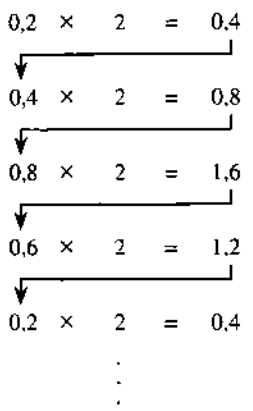


Figura 2.3 • Un número fraccionario puro en base 10 que no tiene su correspondiente forma en el sistema binario de numeración.

Donde $j = i/5^k$ y 5^k debe ser factor de i. Para fracciones decimales de un solo dígito, únicamente 0 y 0,5 dan como resultado binario una fracción exacta (dando solo el 20% de las posibles fracciones decimales de un único dígito); para fracciones de dos dígitos, solo cumplen con la propiedad las fracciones 0,00, 0,25, 0,50 y 0,75, las que corresponden al 4% de las posibles fracciones de dos dígitos, etc. Existe una relación entre los números relativamente primos y las fracciones periódicas, que puede ser útil para entender cuál es la razón por la que algunas fracciones decimales exactas no tienen una forma equivalente binaria que también sea exacta. D. E. Knuth, en *The Art of Computer Programming*, provee algunas profundizaciones sobre el tema.

Representación binaria versus representación decimal

En tanto que la mayoría de las computadoras usan el sistema de numeración binario para la representación interna y el cálculo aritmético, algunas calculadoras y máquinas de oficina utilizan como representación interna el sistema de numeración decimal, con lo que no tienen los problemas de representación mencionados. La causa principal de la utilización del sistema decimal de numeración en las computadoras comerciales no es tanto el problema de la precisión de los números fraccionarios sino el hecho de poder evitar los procesos de conversión decimal-binario en las unidades de entrada y salida, procesos que históricamente requirieron una cantidad de tiempo importante.

Representación de números en los sistemas binario, octal y hexadecimal

Si bien los números binarios reflejan la realidad de la representación interna de los números tal como se utiliza en la inmensa mayoría de las computadoras, tienen como desventaja el hecho de requerir mayor cantidad de dígitos para representar un número dado que cualquier otro sistema de numeración posicional. Asimismo, suele ser más fácil cometer

errores cuando se escriben números binarios debido a la gran cantidad de ceros y unos que hay que utilizar en la representación. En apartados anteriores se ha mencionado a los **sistemas de numeración octal** (sistema de base 8) y **hexadecimal** (de base 16), como sistemas vinculados al sistema de numeración binario. Esta relación está dada por el hecho de ser estas bases potencias de dos, la menor de todas ellas. Se procederá a demostrar que la conversión entre los sistemas de numeración binario, octal y hexadecimal es trivial, y que hay ventajas prácticas significativas en el uso de estos sistemas para la representación de números.

Los números binarios pueden ser considerablemente más grandes (en cantidad de dígitos) que sus equivalentes decimales. Suele resultar práctico como elemento de representación el utilizar aquellos sistemas de numeración cuyas bases son potencia de dos. La conversión entre los sistemas de numeración de bases 2, 8 y 16 es mucho más sencilla que convertir hacia y desde el sistema decimal. Los valores utilizados para los dígitos del sistema octal resultan familiares por cuanto son los primeros ocho dígitos del sistema decimal. En cambio, para el sistema hexadecimal, se requieren seis dígitos más que los que se usan en el sistema decimal. La convención habitual para la representación de los dígitos adicionales (10, 11 ,12, 13, 14, 15) del sistema hexadecimal pasa por el uso de las seis primeras letras del abecedario, sean mayúsculas o minúsculas. La figura 2.4 representa los dígitos utilizados comúnmente en los sistemas de numeración de bases 2, 8, 10 y 16. Al comparar la columna correspondiente al sistema binario con las columnas de los sistemas octal y hexadecimal surge que se requieren tres bits para representar en binario cada uno de los dígitos del sistema octal, y cuatro bits para representar en binario cada uno de los 16 dígitos del sistema hexadecimal. En general, se requieren k bits para representar en binario un dígito del sistema de numeración de base 2^k, siendo k un número entero, por lo que el sistema de numeración de base $8 = 2^3$ requiere tres bits por dígito, en tanto que el sistema de numeración de base $16 = 2^4$ requiere cuatro bits por dígito.

Para convertir un número expresado en el sistema binario al sistema octal, se divide el número binario original en grupos de tres bits cada uno, empezando a partir de la coma decimal, completando el grupo más significativo con ceros, en caso de ser necesario. Luego, cada trío de bits se convierte en forma individual al sistema octal. Para conversiones desde el sistema binario al hexadecimal se utilizan grupos de cuatro bits. Si se pretende convertir el número $(10110)_2$ al sistema de base 8, el procedimiento es el siguiente:

$$(10110)_2 = (010)_2\,(110)_2 = (2)_8\,(6)_8 = (26)_8$$

En el caso de los dos bits de mayor peso, se agregó un cero a la izquierda para completar el trío correspondiente.

Si ahora se considera la conversión del número binario $(10110110)_2$ al sistema hexadecimal, un procedimiento similar al anterior lleva a:

$$(10110110)_2 = (1011)_2\,(0110)_2 = (B)_{16}\,(6)_{16} = (B6)_{16}$$

Binario (base 2)	Octal (base 8)	Decimal (base 10)	Hexadecimal (base 16)
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Figura 2.4 • Valores de los primeros 16 números en los sistemas de numeración binario, octal, decimal y hexadecimal.

Nótese que B, en el sistema de numeración hexadecimal, no es una variable, sino que es el dígito de dicho sistema que representa al número once.

Los métodos de conversión planteados pueden usarse para convertir un número desde cualquier sistema de numeración a cualquier otro sistema, pero es probable que la forma de realizar una conversión, tal como la de convertir $(513{,}03)_6$ a base 7, no resulte demasiado evidente. Como ayuda en estos casos no habituales, se puede realizar la conversión pasando primero por el sistema de numeración decimal, y completando luego la conversión desde el sistema decimal a la base de destino. Como regla general, se utiliza el método polinómico cuando se convierte un número al sistema de numeración decimal, y se utilizan los métodos de multiplicación (para números fraccionarios) y de división y obtención de restos cuando se convierte un número desde el sistema decimal.

2.2.5 Una primera mirada a la aritmética de las computadoras

El análisis detallado del cálculo en computadoras se desarrollará en el capítulo 3. Por el momento se analizará cómo se resuelve la suma binaria, dado que este conocimiento es necesario para la representación de números binarios con signo. La suma binaria se realiza en forma similar a la forma en que se realizan a mano las sumas en el sistema decimal, tal como se ilustra en la figura 2.5. Dos números binarios *A* y *B* se suman de derecha a izquierda, generando un bit de suma y uno de arrastre en cada posición binaria. Dado que los bits menos significativos de *A* y *B* pueden adoptar uno de dos valores, la suma de la columna de las unidades permite plantear solo cuatro posibilidades: 0 + 0, 0 + 1, 1 + 0 y 1+ 1, con un arrastre de 0, como se ve en la figura. El arrastre en la columna de las unidades es siempre nulo. Para las columnas restantes, el arrastre que llega desde la colum-

na anterior puede ser cero o uno, por lo que en cada columna pueden darse hasta ocho combinaciones de entrada, según lo muestra la misma figura 2.5.

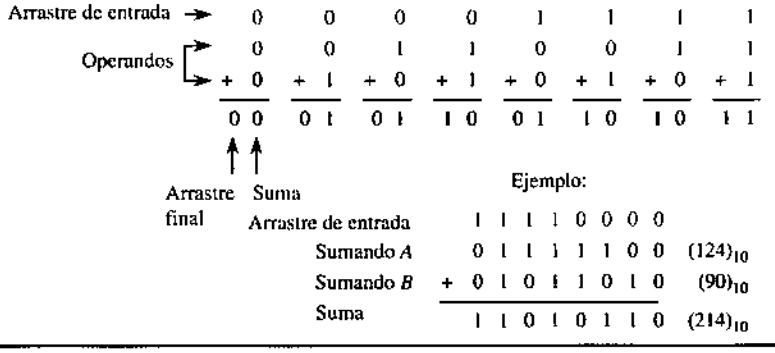


Figura 2.5 • Ejemplo de suma binaria.

Nótese que el mayor número que puede representarse con el formato de ocho bits de la figura 2.5 es $(1111\ 1111)_2 = (255)_{10}$, y que el menor número a representar es $(0000\ 0000)_2 = (0)_{10}$. Los conjuntos de bits 1111 1111 y 0000 0000,* así como todos los patrones de bits intermedios, representan a los números decimales correspondientes al intervalo cerrado que va desde 0 a 255, siendo todos ellos números positivos. Hasta este momento se han considerado solo números sin signo, pero también se hace necesario admitir la representación de números signados, para lo cual se asignará aproximadamente la mitad de los elementos antes mencionados a la representación de los números positivos y la otra mitad a los números negativos. La próxima sección analiza cuatro convenciones de uso habitual en la representación de números binarios signados.

2.2.6 Números signados en formato de punto fijo

Hasta este momento solo se ha considerado la representación de números en formato de punto fijo sin signo. La situación es bastante diferente cuando se pretende representar números signados. Existen cuatro convenciones distintas de uso habitual en la representación de números con signo: magnitud (valor absoluto) y signo, complemento a uno, complemento a dos y notación excedida. Se analizarán todas ellas, de una en una, utilizando como ejemplo la representación de números enteros. Durante el análisis, el lector podrá emplear como referencia la tabla 2.1, que muestra las cuatro representaciones para el caso de un número de tres bits.

* *N. de T.*: Para mayor claridad, los números binarios de más de cuatro bits se separan en grupos de a cuatro (un dígito hexadecimal). Esta separación no influye sobre el valor del número; solo se utiliza para facilitar la lectura.

Decimal	Sin signo	Magnitud y signo	Complemento a 1	Complemento a 2	Exceso 4
7	111	–	–	–	–
6	110	–	–	–	–
5	101	–	–	–	–
4	100	–	–	–	–
3	011	011	011	011	111
2	010	010	010	010	110
1	001	001	001	001	101
+0	000	000	000	000	100
-0	–	100	111	000	100
-1	–	101	110	111	011
-2	–	110	101	110	010
-3	–	111	100	101	001
-4	–	–	–	100	000

Tabla 2.1 • Representaciones de números enteros de tres bits.

Magnitud y signo

La representación en **magnitud y signo** (conocida también como de **valor absoluto y signo**) es la más habitual debido a su uso en el sistema de numeración decimal. Un signo más o un signo menos, colocado a la izquierda del número representado, indica si el número en cuestión es positivo o negativo, tal como se acostumbra al escribir $+12_{10}$ o -12_{10}. En la representación binaria de magnitud y signo se utiliza el bit ubicado más a la izquierda (el de mayor valor absoluto) para representar el signo, asignándosele valor 0 o 1 para representar, respectivamente, el + y el -. Los bits restantes contienen el valor absoluto del número. En esta convención, las representaciones binarias de $+12_{10}$ y -12_{10}, en formato de palabra de ocho bits, se traducen como:

$$+(12)_{10} = (0000\ 1100)_2$$

$$(-12)_{10} = (1000\ 1100)_2$$

El número negativo se obtiene simplemente al reemplazar, a partir de la representación del número positivo, el cero del bit de signo por un uno. Nótese que en esta convención existen dos representaciones para el cero, una positiva y una negativa: 0000 0000 y 1000 0000.

En el formato utilizado en el ejemplo se hace uso de ocho bits, siendo válidas todas las combinaciones de esos ocho bits, por lo que existen $2^8 = 256$ combinaciones diferentes. Sin embargo, esta convención solo puede representar $2^8 - 1 = 255$ combinaciones, dado que +0 y -0 representan el mismo número.

En el desarrollo del texto se hará uso de la representación en magnitud y signo durante el análisis de las representaciones en punto flotante, en la sección 2.3.

Complemento a uno

La operación binaria de **complemento a uno** tiene una resolución trivial: convertir en ceros todos los unos de un número y convertir todos sus ceros en unos. Véase como referencia la cuarta columna de la tabla 2.1. De allí surge que en la representación de complemento a uno, el bit más significativo es 0 para los números positivos y 1 para los negativos, tal como ocurre en la representación de magnitud y signo. El procedimiento de invertir ceros por unos y unos por ceros se conoce como **complementación** de la palabra. Si se vuelve a considerar la representación de $(+12)_{10}$ y $(-12)_{10}$ en formato de ocho bits, ahora usando esta convención de complemento a uno, los resultados serán:

$$(+12)_{10} = (0000\ 1100)_2$$

$$(-12)_{10} = (1111\ 0011)_2$$

Nuevamente, debe notarse la doble representación del cero, mediante 0000 0000 y 1111 1111, respectivamente, para +0 y -0. Como resultado, en esta convención también pueden representarse solo $2^8 - 1 = 255$ valores numéricos a pesar de ofrecer 256 combinaciones.

La representación de complemento a uno no es de uso común. Esto se debe, al menos parcialmente, a la dificultad en la realización de comparaciones derivada de la existencia de dos representaciones para el cero. Ofrece también una complejidad adicional en las operaciones aritméticas como la suma, lo que se analiza en el capítulo 3.

Complemento a dos

El complemento de un número a dos se obtiene en forma similar a la del complemento a uno: tras invertir todos los bits de una palabra, se le suma uno al resultado obtenido, y si esa suma da por resultado un arrastre desde el bit más significativo, el mismo se descarta. Si se analiza la quinta columna de la tabla 2.1, se puede observar que en la representación de **complemento a dos** y signo, el bit más significativo vuelve a ser cero para los números positivos y uno para los negativos. No obstante, esta representación no ofrece la característica desafortunada vista en los dos modelos anteriores, dado que solo posee una única representación para el cero. Para ver que esta afirmación es cierta, considérese la obtención del valor negativo correspondiente al $(0)_{10}$, cuyo formato es:

$$(0)_{10} = (0000\ 0000)_2$$

El cálculo del complemento a uno de esta palabra lleva a obtener $(1111\ 1111)_2$, y si a este valor se le suma 1, en formato de ocho bits se obtiene $(0000\ 0000)_2$. Así, la representación del $(-0)_{10} = (0000\ 0000)_2$. El arrastre proveniente del bit más significativo siempre se descarta en las operaciones de suma en complemento a dos, excepto cuando se detecta una situación de desborde. Dado que hay una única representación para el cero, y dado que todas las combinaciones son válidas, existen $2^8 = 256$ números diferentes a ser representados en esta convención.

Si se vuelve a considerar la representación de los números decimales +12 y -12, en formato de ocho bits, ahora con la representación de complemento a dos y signo, el procedimiento requerirá comenzar con $(+12)_{10} = (0000\ 1100)_2$. Si se invierte o niega el valor, se obtendrá $(1111\ 0011)_2$. Si al resultado se le suma uno, se obtiene $(1111\ 0100)_2$, por lo que $(-12)_{10} = (1111\ 0100)_2$.

$$(+12)_{10} = (0000\ 1100)_2$$

$$(-12)_{10} = (1111\ 0100)_2$$

Si se considera que el cero es positivo, lo que es razonable dado que su bit de signo es cero, existen en la convención igual cantidad de números positivos y negativos. Los números positivos arrancan desde cero, pero los negativos se inician en -1, por lo que el valor absoluto del número más negativo es mayor, en una unidad, que el valor absoluto del número más positivo. El mayor número positivo que se puede expresar en esta convención es +127 y el menor número negativo (el número negativo de mayor valor absoluto) es -128. No existe, por consiguiente, número positivo alguno a ser representado que pueda corresponderse con el complemento de -128. En efecto, si se trata de obtener el complemento a dos de -128, se volverá a obtener un número negativo, según puede observarse a continuación:

$$(-128)_{10} = (1000\ 0000)_2$$

$$0111\ 1111 + 1 = (1000\ 0000)_2$$

La convención de complemento a dos y signo es la representación más utilizada en computadoras, y es la que se empleará a lo largo de este texto.

Representación excedida (desplazada)

En la representación con **exceso** o **desplazamiento**, los números se tratan como si no tuvieran signo, pero se los "desplaza" en su valor por medio de la resta de otro número conocido como exceso o desplazamiento. La idea es asignar el valor numérico más pequeño, formado por todos ceros, al valor negativo del desplazamiento, y asignar los valores restantes en secuencia a medida que los patrones binarios aumentan en magnitud. Una for-

ma conveniente de pensar en una representación excedida es la de imaginar al número como representado por la suma de su expresión complemento a dos y otro número, al que se designa como "exceso" o "desplazamiento". Nuevamente, puede emplearse como referencia la tabla 2.1, en este caso la columna de la derecha, para ejemplos de aplicación.

Si se vuelve a plantear el ejemplo de la representación de $(+12)_{10}$ y $(-12)_{10}$ en formato de ocho bits pero utilizando ahora una representación en exceso 128, los dos números se obtendrán sumando 128 al número original y determinando luego la versión binaria sin signo. Para $(+12)_{10}$ se calculará $(128 + 12 = 140)_{10}$, lo que llevará al patrón $(1000\ 1100)_2$. Para $(-12)_{10}$, el cálculo a realizar es $(128 - 12 = 116)_{10}$, obteniéndose ahora la representación binaria $(0111\ 0100)_2$:

$$(+12)_{10} = (1000\ 1100)_2$$

$$(-12)_{10} = (0111\ 0100)_2$$

Nótese que no hay ningún significado numérico asociado con el valor del exceso; su efecto es simplemente el de desplazar la representación de los números expresados en complemento a dos.

Hay una sola representación desplazada para el 0, dado que la representación excedida constituye simplemente una versión desplazada de la representación complemento a dos. En el caso anterior, el valor del exceso fue elegido para que tuviese el mismo formato que el número negativo más grande, lo que produce como efecto que los números aparezcan ordenados numéricamente si se los mira en una representación binaria no signada. Así, el número más negativo es $(-128)_{10} = (0000\ 0000)_2$, y el más positivo es $(+127)_{10} = (1111\ 1111)_2$. Esta representación simplifica las comparaciones entre números, dado que las representaciones binarias de los números negativos tienen valores numéricamente menores que las representaciones de los números positivos. Esto se hace importante cuando se representan los exponentes de los números en punto flotante. En este tipo de representación se requiere comparar dichos exponentes de dos cantidades para igualarlos, en caso de ser necesario, para sumar o restar. Las representaciones de punto flotante se analizarán en la sección 2.3.

2.2.7 Decimal codificado en binario

Un número cualquiera puede representarse en su expresión de base 10 mediante codificación binaria. Cada dígito del sistema de numeración decimal se representa con cuatro bits, dando lugar a lo que se conoce como **decimal codificado en binario** (BCD, *binary coded decimal*). Cada dígito BCD puede tomar uno de diez valores. Dado que para cada dígito decimal hay $2^4 = 16$ posibles combinaciones binarias, resulta que quedan seis combinaciones binarias de cuatro bits sin utilizar. En el ejemplo de la figura 2.6 se presenta un número de cuatro dígitos decimales significativos, lo que implica la existencia de $10^4 = 10.000$ combinaciones binarias válidas, de las $2^{16} = 65.536$ combinaciones que pueden formarse con 16 bits.

Si bien algunas de las combinaciones binarias no se utilizan, el formato BCD se emplea comúnmente en calculadoras y en aplicaciones comerciales. Existen menos problemas cuando se representan fracciones decimales exactas en este formato, en contraposición con la representación binaria. No hay necesidad de convertir la información que se ingresa en formato decimal (como en una calculadora) o de convertirlos desde un formato de representación binaria a su correspondiente expresión decimal.

La realización de operaciones aritméticas con números BCD signados puede no resultar obvia. Si bien el uso de representaciones de magnitud y signo es habitual en el sistema de numeración decimal, la computadora utiliza un método diferente para la representación de números decimales signados. En la representación de **complemento a nueve**, los números positivos se representan en la forma BCD habitual, pero el dígito decimal más significativo adopta un valor menor que 5 si el número es positivo, y 5 o más si el número representado es negativo. El complemento a nueve se obtiene restando cada dígito de 9. Por ejemplo, el número decimal +301 se representa como 0301 (o, simplemente, como 301) tanto en las representaciones de complemento a nueve y a diez, según se muestra en la figura 2.6a. El número negativo, en complemento a nueve, es 9698 (figura 2.6b), y se obtiene restando de 9 cada uno de los dígitos del 0301.

El **complemento a diez** se obtiene sumando uno al complemento a nueve, por lo que la representación de -301 en notación de complemento a diez será 9698 + 1 = 9699, tal como se ilustra en la figura 2.6c. En este modelo, los números positivos asumen un rango de 0 a 4999, en tanto que los números negativos están en el rango de 5000 a 9999.

(a)	0000	0011	0000	0001	$(+301)_{10}$	Complemento a 9 y a 10
	$(0)_{10}$	$(3)_{10}$	$(0)_{10}$	$(1)_{10}$		
(b)	1001	0110	1001	1000	$(-301)_{10}$	Complemento a 9
	$(9)_{10}$	$(6)_{10}$	$(9)_{10}$	$(8)_{10}$		
(c)	1001	0110	1001	1001	$(-301)_{10}$	Complemento a 10
	$(9)_{10}$	$(6)_{10}$	$(9)_{10}$	$(9)_{10}$		

Figura 2.6 • Representación BCD de los números 301 y -301 en (a) complemento a nueve, (b) complemento a diez.

2.3 Formato de representación en punto flotante

La representación de números en formato de punto fijo, que fuera analizada en la sección 2.2, ubica la coma decimal en una posición fija, y una cantidad fija y determinada de dígitos tanto a la izquierda como a la derecha de la coma decimal. Por consiguiente, este tipo de representación puede requerir una gran cantidad de dígitos para representar un rango de números apropiado para determinada aplicación. Por ejemplo, una computadora que deba representar

números del orden del billón (millón de millones)[2] requerirá, al menos, 40 bits a la izquierda de la coma decimal, dado que 10^{12} es aproximadamente igual a 2^{40}. Si se debiera representar además una fracción equivalente al billonésimo, se requerirían adicionalmente otros 40 bits a la derecha de la coma decimal, lo que daría por resultado una palabra de 80 bits.

En la práctica, suelen aparecer en los cálculos números que pueden ser mucho mayores o mucho menores que los mencionados, lo que requiere aún más espacio de almacenamiento en la computadora. Para manejar y almacenar números con 80 o más bits de precisión se requiere una buena cantidad de hardware, y, por otra parte, las operaciones de cálculo pueden llegar a resolverse más lentamente cuando se trabaja con cantidades mayores de bits. No obstante, también es cierto que en muchas ocasiones no se requiere una precisión muy fina cuando se trabaja con cantidades muy grandes y que, por el contrario, no se requieren números muy grandes cuando se trabaja con operandos pequeños. Como conclusión, la computadora más eficiente puede imaginarse como aquello que solo tiene la precisión que se requiere.

2.3.1 Rango y precisión en números de punto flotante

La representación de números en **formato de punto flotante** permite representar un amplio rango de números con poca cantidad de dígitos binarios; para esto, se separan los dígitos utilizados para determinar la precisión de la representación de aquellos necesarios para representar el rango. El número decimal de formato de punto flotante utilizado para expresar el número de Avogadro es

$$+6{,}023 \times 10^{23}$$

En esta representación, el rango se expresa a través de una potencia de la base 10, 10^{23} en este caso, y la precisión se expresa a través del número de punto fijo, 6,023 en este caso. En la representación de números de formato de punto flotante, la parte que corresponde al valor en punto fijo se conoce habitualmente como **mantisa** del número. Así, la representación de punto flotante de una cantidad numérica queda determinada por un trío de elementos numéricos: el signo, el exponente y la mantisa.

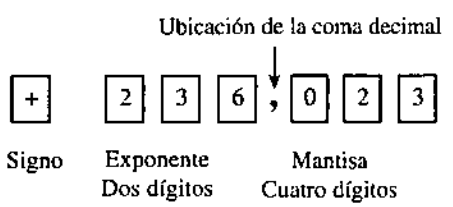


Figura 2.7 • Representación de un número decimal en formato de punto flotante.

2. Debe tenerse en cuenta que en la denominación norteamericana de los números, 10^{12}, aquí mencionado como "billón", se denomina "trillón". Para los norteamericanos, el "billón" corresponde a lo que en otras partes del mundo se conoce como "mil millones" (10^9).

El rango de la representación queda determinado básicamente por la cantidad de dígitos del exponente (en este ejemplo se utilizan dos dígitos para la representación del exponente) y la base a la que ese exponente afecta (diez, en este caso). La precisión queda determinada por la cantidad de dígitos de la mantisa (cuatro, en este ejemplo). Así, el número completo puede representarse a través de su signo y 6 dígitos, dos para el exponente y cuatro para la mantisa. La figura 2.7 muestra un formato que podría utilizarse dentro de una computadora para la representación del número así expresado. Nótese que, en este ejemplo, los dígitos se empaquetan colocando primero el signo, luego el exponente y por último la mantisa. Este ordenamiento resulta práctico en las operaciones de comparación de cantidades expresadas en el formato de punto flotante. El lector debe tener en cuenta que la coma decimal no debe almacenarse dado que ocupa siempre la misma posición dentro de la mantisa, la que está representada en formato de punto fijo. (Esta idea se analiza en la sección 2.3.2.)

Si se requiere un rango mayor, y si a cambio se está dispuesto a sacrificar precisión, se pueden usar tres dígitos para la parte fraccionaria y dejar entonces tres dígitos disponibles para el exponente, sin necesidad de aumentar la cantidad de dígitos totales de la representación. Un método alternativo para aumentar el rango es el de aumentar la base del sistema de representación, lo que incrementa la precisión de los números más chicos mientras disminuye la de los números más grandes. La posibilidad de plantear soluciones de compromiso entre rango y precisión es una de las ventajas principales de la representación en formato de punto flotante, aun cuando reducir la precisión puede provocar problemas que llevan al desastre, tal como se lo describe en un ejemplo planteado en la sección 2.4.

2.3.2 La normalización y el esquema de bits implícitos

La representación de números en formato de punto flotante presenta como eventual problema el hecho de que un mismo número puede representarse de distintas maneras, lo que complica las comparaciones y las operaciones aritméticas. Por caso, las siguientes formas numéricas son todas equivalentes:

$$3584{,}1 \times 10^0 = 3{,}5841 \times 10^3 = 0{,}35841 \times 10^4$$

Con el objeto de evitar el uso de representaciones múltiples para el mismo número, las representaciones de números en formato de punto flotante trabajan de manera **normalizada**. En este concepto, la coma decimal se desplaza a derecha o a izquierda y se ajusta el exponente en forma coherente con el desplazamiento de la coma decimal hasta ubicarla a la izquierda del dígito no nulo más significativo. Por consiguiente, la expresión normalizada del número que aparece en el ejemplo anterior es la tercera, a la derecha de las tres. Como inconveniente, este esquema no permite representar el cero, por lo que su representación debe hacerse a través de un procedimiento de excepción. La excepción

planteada consiste en la representación del cero a través de un número con mantisa totalmente nula.

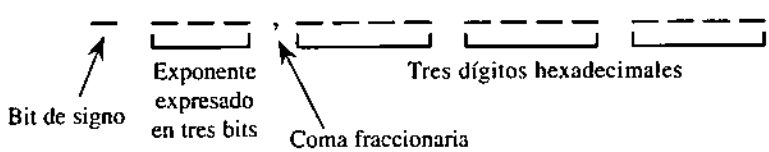
Si la mantisa se representa como un número binario (esto es, en el sistema binario de numeración), y si la condición de normalización consiste en que la mantisa normalizada siempre comience con un 1, no hay necesidad de almacenar ese 1, por lo que, efectivamente, la mayoría de los métodos de representación de números en formato de punto flotante proceden a no almacenar dicho bit inicial. En vez de almacenarlo, lo que se hace es "recortarlo" antes de empaquetar el número para su almacenamiento, recuperándolo al desempaquetar y llevar el número a su representación de mantisa y exponente. Como resultado de esta operación se obtiene lugar para un bit adicional a la derecha de la mantisa, lo que mejora la precisión de la representación. El bit así eliminado se suele denominar **bit implícito.** Por ejemplo, si en un formato determinado la mantisa luego de la normalización se representa como 0,11010, el patrón a ser almacenado es 1010 y el bit más significativo se trunca, se esconde, se sobreentiende. En el análisis de la norma de representación de punto flotante IEEE[3] 754 veremos que la misma representa números utilizando un bit implícito.

2.3.3 Representación de números de punto flotante dentro de la computadora. Introducción

Para ilustrar las características importantes de la representación de números en formato de punto flotante se procederá a definir un formato simple de representación. En principio, este formato puede parecer innecesariamente complicado. La mantisa se representará en formato de magnitud y signo, con un único bit para representar el signo y tres dígitos hexadecimales (doce bits) como tamaño de la representación. El exponente será un número de tres bits, expresado en exceso 4, con base 16. La forma normalizada de la representación tiene la coma fraccionaria ubicada a la izquierda de los tres dígitos hexadecimales.

Los bits que forman la palabra se empaquetarán de modo tal que el bit de signo quede a la izquierda, seguido por el exponente de tres bits y, a continuación, los tres dígitos hexadecimales de la mantisa. El formato analizado no almacena ni la base del sistema de representación ni la coma fraccionaria.

La razón fundamental para utilizar este formato aparentemente extraño es la posibilidad que ofrece para comparar directamente dos números por =, ≠, ≥ y ≤ en su forma empaquetada, lo que se muestra en la ilustración siguiente:



3. IEEE: Institute of Electrical and Electronics Engineers.

En este esquema, la representación del número decimal 358 requiere como primer paso la conversión del número expresado en formato de punto fijo a su representación de punto flotante en la base de destino. Utilizando los métodos de conversión planteados en la sección 2.2.4, la conversión del número decimal al sistema hexadecimal da por resultado:

		Entero	Resto
358/16	=	22	6
22/16	=	1	6
1/16	=	0	1

Lo que lleva a afirmar que $(358)_{10} = (166)_{16}$. El paso siguiente consiste en la conversión del número desde su formato de punto fijo al de punto flotante:

$$(166)_{16} = (166,0)_{16} \times 16^0$$

Nótese que la forma 16^0 refleja la base 16 elevada a potencia 0, y que el número 16 escrito como lo está en la expresión anterior hace pensar en su representación decimal. Esto es, para ser correctos, $(16^0)_{10} = (10^0)_{16}$. Este juego de representaciones solo se muestra como una notación conveniente cuando se describe un formato de punto flotante. El paso siguiente consiste en la normalización del número representado:

$$(166,0)_{16} \times 16^0 = (0,166)_{16} \times 16^3$$

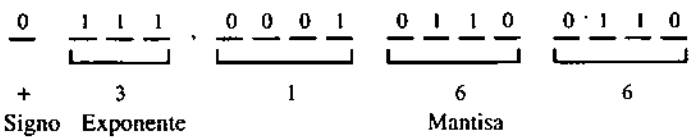
Finalmente, el último paso consiste en completar los campos numéricos de la representación. El valor a representar es positivo, por lo que en la posición del bit de signo corresponde un cero. El exponente es 3, pero habiendo definido que se representa en exceso 4, corresponde representarlo según la operación siguiente:

	0 1 1	$(+3)_{10}$
Exceso 4	+ 1 0 0	$(+4)_{10}$
Exponente en exceso 4	1 1 1	

Obviamente, se podría haber planteado directamente 3 + 4 = 7 en decimal, procediendo luego a la conversión del resultado decimal al sistema binario:

$$(7)_{10} = (111)_2$$

Por último, se expresa cada uno de los dígitos hexadecimales en binario, siendo 1 = 0001, 6 = 0110 y el otro 6 = 0110. El formato final de la representación es el siguiente:



Debe notarse nuevamente que la coma fraccionaria no se representa en forma explícita en el patrón de bits planteado, sino que su presencia está implícita. Los espacios entre dígitos se han introducido solo por cuestiones de claridad y no deben interpretarse de manera alguna como que los bits se almacenan con espacios intermedios. En realidad, el formato almacenado en la memoria de la computadora se asemeja al que sigue:

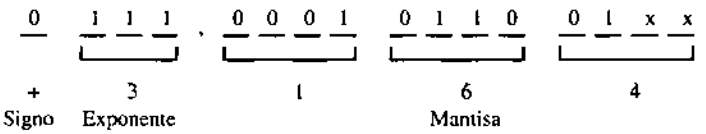
0111 0001 0110 0110

El uso de la representación de exceso 4 para el exponente en reemplazo de una representación de complemento a dos o de magnitud y signo simplifica las operaciones de suma y resta de números en formato de punto flotante (tema que se tratará en detalle en el capítulo 3). Para sumar o restar dos números normalizados en formato de punto flotante, el menor de los dos exponentes (menor en grado, no en magnitud) debe incrementarse primero hasta alcanzar el valor del mayor de los dos exponentes. Esto mantiene el rango y produce además el efecto de denormalizar el menor de los dos operandos. Con el objeto de determinar cuál de los dos exponentes es el mayor, solo es necesario tratar los dos patrones de bits como si fuesen números sin signo y realizar la comparación en esas condiciones. Esto significa que, si la representación de los exponentes se realiza en exceso 4, el menor de los exponentes es -4, el que se representa como 000. El mayor exponente que puede representarse es +3, que se representa como 111. Los patrones correspondientes a los restantes exponentes, -3, -2, -1, 0, +1 y +2, aparecen ordenados como 001, 010, 011, 100, 101 y 110.

Queda claro ahora que ante el patrón de bits utilizado para representar el número $(358)_{10}$, todo lo que hace falta para identificar el número representado es conocer la descripción de la representación empleada. El bit de signo vale cero, lo que significa que el número es positivo. El exponente, en forma no signada, es el número decimal 7, lo que implica, dado que la representación está dada en exceso 4, que al restar ese exceso 4 del valor de 7 se obtendrá el valor real del exponente, o sea 3. La mantisa está agrupada como un conjunto de tres dígitos hexadecimales, lo que permite recuperar una mantisa fraccionaria de $(0,166)_{16}$. La recolección de todos los valores termina recuperando el número representado $(+0,166 \times 16^3)_{16} = (358)_{10}$.

Cabría preguntarse cómo se modificaría la representación del número analizado si se admitiesen 10 bits en la representación de la parte fraccionaria, en lugar de los 12 bits hasta ahora utilizados para formar los tres dígitos hexadecimales. Una aproxima-

ción a la respuesta puede consistir en redondear la fracción y ajustar el exponente según resulte necesario. Otra aproximación, que se utiliza en este caso, es simplemente la de **truncar** los bits menos significativos, evitando así la necesidad de maniobras que impliquen ajustes en el exponente. En consecuencia, el número que en realidad se representa es el



Si los bits eliminados fueran tratados como ceros, el patrón de ceros y unos indicados representaría el número $(0,164 \times 16^3)_{16}$. Este criterio, por el cual se truncan bits, produce un error polarizado debido a que los valores de 00, 01, 10 y 11 en esos bits se tratan todos como si fuesen cero, por lo que el error se encuentra en el rango que va desde 0 hasta $(0,003)_{16}$. La polarización del error se refiere a que el error no es simétrico respecto del cero. Un análisis más detallado de las consecuencias de este problema, que no será tratado aquí con mayor profundidad, fue desarrollado por V. C. Hamacher y otros en *Computer Organization.*

Nuevamente, es importante insistir en que, no importa cuál sea el formato de punto flotante de que se trate, este debe ser conocido por todo aquel que pretenda almacenar o recuperar números que se hallan representados en ese formato. El IEEE es quien ha tomado el liderazgo en la normalización de formatos de punto flotante. La norma IEEE 754 para la representación de números en formato de punto flotante, casi de uso universal, se analiza en la sección 2.3.5.

2.3.4 Errores en la representación de punto flotante

El hecho de que la precisión finita introduce errores implica que debería considerarse cuán grande es el error (por error se entiende la distancia entre dos números representables consecutivos) y si es razonable para la aplicación en uso. Como ejemplo de una situación crítica, considérese el número un millón, representado en formato de punto flotante, al que se le restan consecutivamente un millón de unos. Si el error de la representación es mayor que 1, el resultado de esa operación podría dar otra vez un millón.[4]

Con el objeto de caracterizar el error, el rango y la precisión, se utilizarán las siguientes nomenclaturas:

4. La mayoría de las computadoras actuales permiten que este límite superior valga, como mínimo, 8 millones cuando se utiliza la precisión convencional.

b	Base
s	Cantidad de dígitos significativos (no bits) en la mantisa
M	Mayor exponente
m	menor exponente

La cantidad de dígitos significativos de la mantisa se representa por s, que es distinto de la cantidad de bits de dicha mantisa si la base es distinta de 2 (dado que, por ejemplo, usar base 16 implica usar cuatro bits por cada dígito). En general, si la base es 2^k, siendo k un entero, se utilizan k bits para representar cada dígito. El uso de un 1 implícito incrementa s en 1 aun cuando no incrementa la cantidad de números que pueden representarse. En el ejemplo anterior se tienen tres dígitos significativos en base 16, y 12 bits que conforman esos tres dígitos hexadecimales. Existen tres bits en el exponente representado en exceso 4, lo que le asigna al exponente un rango que va desde -2^2 hasta $2^2 - 1$. Para este caso, $b = 16$, $s = 3$, $M = 3$ y $m = -4$.

Al analizar una representación de punto flotante deben considerarse cinco características importantes: cuál es la cantidad de números que permite representar, cuáles son los números de mayor y menor magnitud (fuera del cero), y cuáles los tamaños de la mayor y de la menor diferencia entre números consecutivos.

La cantidad de números representables puede determinarse según lo ilustrado en la figura 2.8. El bit de signo puede adoptar dos valores, como se indica en la posición A. La cantidad total de exponentes a representar se indica en la posición B. Debe tenerse en cuenta que no todas las posibles combinaciones de bits se pueden considerar exponentes válidos. La norma de punto flotante IEEE 754, que se analizará en breve, ofrece un exponente mínimo de -126 aun cuando el campo correspondiente, de ocho bits, podría manejar un número tan chico como -128. Los exponentes prohibidos se reservan para números especiales, tales como cero e infinito.

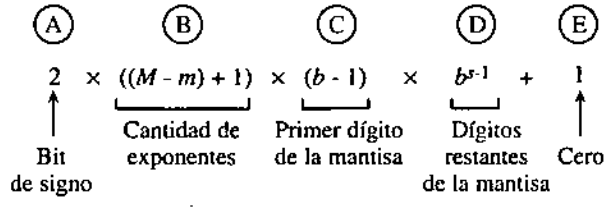


Figura 2.8 • Determinación de la cantidad de valores representables en punto flotante.

Se considera a continuación el primer dígito de la mantisa, el que, en un formato normalizado, puede adoptar cualquier valor excepto el cero (salvo cuando se utilice un formato con 1 implícito). Este dígito se indica en la posición C. Los dígitos restantes pueden adoptar cualquiera de los b valores diferentes del sistema de numeración utilizado, lo que

se indica con b^{s-1} en la posición D. Si se utiliza un bit implícito, debe eliminarse la posición C, lo que a su vez hace que la posición D sea reemplazada por b^s. Finalmente, se requiere una representación para el cero, la que se indica en la posición E.

Se analizarán ahora los números extremos de la representación planteada. El mínimo valor que puede representarse corresponde al número con el menor exponente y la menor mantisa normalizada no nula. Debe haber un valor no nulo en el primer dígito, y dado que 1 es el menor valor que se puede colocar en ese lugar, el menor valor de la mantisa es b^{-1}. Por consiguiente, el número de menor magnitud es $b^m \cdot b^{-1} = b^{m-1}$. En forma similar, el máximo número a representar corresponde a aquel que tenga la mayor mantisa (aquella que tenga todos sus bits en 1) y el máximo exponente, lo que resulta equivalente a $b^M \cdot (1 - b^{-s})$.

Para el cálculo de las diferencias máxima y mínima se procede en forma semejante. La menor diferencia se produce cuando el exponente se acomoda a su valor mínimo y ocurre un cambio en el bit menos significativo de la mantisa. Este salto tiene como valor $b^m \cdot b^{-s} = b^{m-s}$. La diferencia más grande se produce cuando el exponente adopta su máximo valor y ocurre un cambio en el bit menos significativo de la mantisa. En este caso, la diferencia vale $b^M \cdot b^{-s} = b^{M-s}$.

Como ejemplo, considérese una representación de punto flotante en la que se tiene un bit de signo, un exponente de dos bits en notación exceso 2 y una mantisa normalizada binaria, de tres bits, con el primer uno visible, no implícito. La representación del cero corresponde a la combinación 00 0000. La figura 2.9 representa, sobre una recta numérica, todas las posibles representaciones válidas en este formato. Nótese la existencia de un salto apreciablemente grande entre el 0 y el primer número representable, debido a que las representaciones normalizadas no admiten formatos que correspondan a valores entre cero y el primer valor representable.

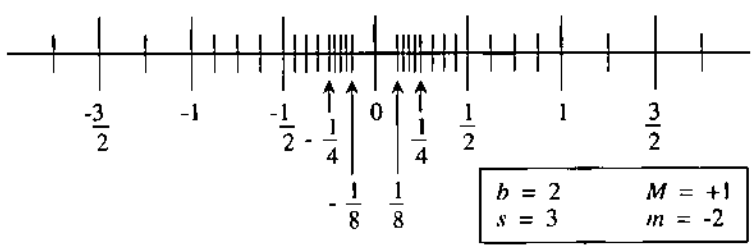


Figura 2.9 • Recta numérica y las posibles representaciones de números en un formato sencillo de punto flotante.

El mínimo número representable corresponde al caso en el que tanto el exponente como la mantisa adoptan sus valores mínimos. El mínimo valor del exponente es -2 y la mínima mantisa normalizada es $(0,100)_2$. En consecuencia, el mínimo número que puede representarse es

$$b^m \times b^{-1} = b^{m-1} = 2^{-2-1} = 1/8$$

En forma similar, el número máximo a representar se obtiene cuando tanto la mantisa como el exponente adoptan sus valores máximos. La mayor de las mantisas es aquella en la cual todos sus bits son 1, lo que responde a un valor menor que 1 en 2^{-3}, debido a que se han definido tres dígitos para la mantisa. Por consiguiente, el máximo número a representar es

$$b^M \times (1 - b^{-s}) = 2^1 \times (1 - 2^{-3}) = 7/4$$

La distancia más chica entre representaciones ocurre cuando el exponente adopta su valor mínimo y se produce un cambio en el bit menos significativo de la mantisa, siendo su valor

$$b^m \times b^{-s} = b^{m-s} = 2^{-2-3} = 1/32$$

Análogamente, la distancia máxima se produce cuando se tiene el máximo valor del exponente y se altera el bit menos significativo de la mantisa, lo que da

$$b^M \times b^{-s} = b^{M-s} = 2^{1-3} = 1/4$$

La cantidad de combinaciones asignadas a la representación de números validos es menor que la cantidad total de combinaciones, debido a la normalización. Tal como se analizara anteriormente, la cantidad de números representables está determinada por cinco elementos, que toman en cuenta el bit de signo, los exponentes, el primer dígito significativo, los demás dígitos y el formato adoptado para la representación del cero. El cálculo a realizar es el siguiente:

$$2 \times ((M - m) + 1) \times (b - 1) \times b^{s-1} + 1$$

$$= 2 \times ((1 - (-2)) + 1) \times (2 - 1) - 2^{3-1} + 1 = 33$$

Debe notarse que los intervalos son pequeños para números pequeños y se hacen mayores para números grandes. De hecho, el error relativo es, aproximadamente, el mismo a lo largo de todo el rango. Si se toma la relación entre un intervalo grande y un número grande, y se compara con la relación entre un intervalo pequeño y un número también pequeño, las relaciones son las mismas:

$$\text{Un intervalo grande} \longrightarrow \frac{b^{M-s}}{b^M (1 - b^{-s})} = \frac{b^{-s}}{1 - b^{-s}} = \frac{1}{b^s - 1}$$
$$\text{Un número grande} \longrightarrow$$

y

$$\text{Un intervalo pequeño} \longrightarrow \frac{b^{m-s}}{b^m (1 - b^{-s})} = \frac{b^{-s}}{1 - b^{-s}} = \frac{1}{b^s - 1}$$
$$\text{Un número pequeño} \longrightarrow$$

Se utiliza en este caso la representación de un número pequeño, en lugar de la representación correspondiente al mínimo número admisible, debido a que el intervalo, grande, entre cero y el primer número representable, es un caso especial.

Ejemplo

Se propone considerar el problema de la conversión del número 9,375 x 10^{-2} a un formato de notación científica utilizando el sistema binario de numeración. Esto significa que el resultado debería adoptar la forma *x,yy* x 2^{E}. El proceso se inicia con la conversión del valor decimal punto flotante a un formato de punto fijo. Con este objeto se desplaza la coma decimal dos posiciones a la izquierda, lo que corresponde al exponente -2. El número resultante es 0,09375. Mediante el método de las multiplicaciones se procede a convertir el número decimal así obtenido al sistema binario, representándolo en punto fijo:

0,09375	x	2	=	0,1875
0,1875	x	2	=	0,375
0,375	x	2	=	0,75
0,75	x	2	=	1,5
0,5	x	2	=	1,0

Entonces, resulta que $(0,09375)_{10} = (0,00011)_2$. El último paso consiste en la conversión del valor a la representación normalizada de punto flotante, lo que lleva a que 0,00011 = 0,00011 x 2^0 = 1,1 x 2^{-4}. •

2.3.5 La norma de representación IEEE 754

Existen muchas maneras de representar números en formato de punto flotante, algunas de las cuales ya han sido analizadas. Cada representación tiene sus características propias en términos de rango, precisión y cantidad de elementos que pueden representarse. En un esfuerzo por mejorar la portabilidad de los programas y asegurar la uniformidad en la exactitud de las operaciones en el formato de punto flotante, el IEEE desarrolló su norma IEEE 754 para la representación de números en el formato de punto flotante. Existen algunas líneas de productos anteriores a dicha norma que no la utilizan, como las computadoras IBM/370, las computadoras VAX (DEC, *Digital Equipment Corporation*) y la línea Cray, pero prácticamente todas las nuevas arquitecturas utilizan en alguna forma la estructura IEEE 754.

La norma IEEE 754, tal como se la describe, debe estar asociada con un sistema de computación, no siendo necesario que la relación sea exclusivamente con su hardware. En consecuencia, mientras la computadora siga respetando la norma, el soporte puede realizarse a través de una combinación de hardware y software.

2.3.5.1 Formatos

Hay dos formatos principales en la norma IEEE 754: el formato de **simple precisión** y el formato de **doble precisión**. La figura 2.10 resume la distribución de ambos formatos. El formato de simple precisión requiere 32 bits, mientras que el de doble precisión utiliza 64 bits. El formato de doble precisión es, simplemente, una versión más amplia del formato de simple precisión.

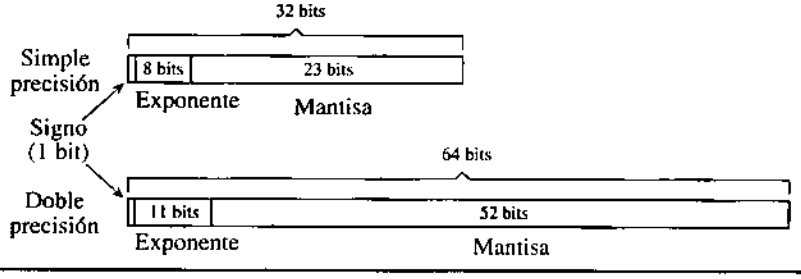


Figura 2.10 • Formatos IEEE 754 para punto flotante simple precisión y doble precisión.

El bit de signo se ubica en la posición del bit más significativo, a la izquierda del número, e indica que el número a representar es positivo o negativo mediante un 0 o un 1, respectivamente. A continuación se ubica el exponente, de ocho bits, en formato exceso 127 (no 128). Tal como se describe luego, las combinaciones 0000 0000 y 1111 1111 se reservan para casos especiales. Para el formato de doble precisión, se utiliza un exponente de 11 bits, expresado en exceso 1023, y se reservan las combinaciones 000 0000 0000 y 111 1111 1111. A continuación, se representa la mantisa, en binario y formada por 23 bits. A la izquierda de la coma fraccionaria existe un bit implícito, el que cuando se lo considera en conjunto con la mantisa de simple precisión, constituye una palabra de 24 bits, de la forma 1,fffff...f. En este formato, el patrón fffff..f representa los 23 bits de la mantisa que se almacenan. El formato de doble precisión también utiliza un bit implícito a la izquierda de la coma fraccionaria, lo que representa un valor significativo de 53 bits. En ambos formatos, el número se representa **normalizado** a menos que la estructura soporte números **denormalizados**, según se describirá más adelante.

Existen en la norma cinco tipos de números que se representan. Los números no nulos adoptan los formatos descriptos anteriormente. La norma admite una llamada representación "limpia" del cero, formada por la combinación reservada 0000 0000 en el campo del exponente, y todos ceros en la mantisa. El bit de signo puede ser cero o uno, por lo que hay dos representaciones para el cero.

El infinito puede representarse mediante la combinación reservada 1111 1111 en el exponente, acompañada por una mantisa de valor cero y el bit de signo en cero o uno. La representación del infinito se usa para manejar situaciones de desborde o para ofrecer una representación válida para la división de un número (distinto de cero) por cero. Si se plantea la cuestión de dividir cero por cero o la de dividir infinito por infinito, el resultado

queda indeterminado. En este caso, la representación **NaN** (*not a number*: no es numérico) presenta en el campo del exponente la combinación reservada 1111 1111, en la mantisa un valor no nulo, y el bit de signo indistintamente en 0 o 1. Puede obtenerse también un NaN si se intenta calcular la raíz cuadrada de -1.

Tal como sucede en todas las representaciones normalizadas, existe un intervalo importante entre cero y el primer número que puede representarse. La representación de números incluidos en este intervalo se resuelve mediante la representación del "cero sucio", denormalizado. En este caso, el bit de signo puede ser 0 o 1, el campo del exponente contiene la combinación reservada 0000 0000, que representa -126 en el caso de simple precisión y -1022 para doble precisión, en tanto que la mantisa contiene el valor real correspondiente a la magnitud del número. Por lo tanto, no hay bit implícito en este formato. Nótese que la notación *denormalizada* no es una representación *no normalizada*. La diferencia clave entre ambas palabras consiste en que para un número dado existe una sola representación denormalizada, en tanto que existen muchas representaciones no normalizadas.

La figura 2.11 ilustra algunos ejemplos de números expresados en formato de punto flotante según la norma IEEE 754. Los ejemplos desde (a) hasta (h) se representan en formato de simple precisión y el ejemplo (i) se muestra en formato de doble precisión. El ejemplo (a) ilustra un número cualquiera en formato de simple precisión. Debe notarse que la mantisa es 1,101, y que, sin embargo, solo se representa explícitamente la fracción (101). El ejemplo (b) utiliza el mínimo exponente en simple precisión (-126) y el ejemplo (c) utiliza el exponente máximo en simple precisión (+127).

Los ejemplos (d) y (e) ilustran las dos representaciones del cero. El ejemplo (f) representa el formato del $+\infty$. Existe, asimismo, un formato alternativo para $-\infty$. El ejemplo (g) muestra un número denormalizado. Nótese que si bien el número en sí es 2^{-128}, el exponente mínimo representable sigue siendo -126. El exponente para números denormalizados en simple precisión siempre es -126, representado por la combinación 0000 0000 y una mantisa no nula. La mantisa representa directamente la magnitud del número. Así se tiene que $+2^{-128} = +0{,}01 \times 2^{-126}$, representada por la combinación indicada en la figura 2.11g.

El ejemplo (h) muestra un NaN de simple precisión. Un NaN puede ser positivo o negativo. Finalmente, el ejemplo (i) replantea la representación de 2^{-128}, pero utilizando ahora el formato de doble precisión. La representación corresponde a un número cualquiera de doble precisión y, por ende, no hay consideraciones especiales para hacer en este caso. Nótese que 2^{-128} tiene una mantisa de 1,0, por lo que la parte fraccionaria está totalmente formada por ceros.

Además de los formatos de simple y doble precisión, existen dos formatos que se conocen como **extendido simple** y **extendido doble**. Los formatos extendidos no son visibles para el usuario, pero se usan con el objeto de mantener internamente una mayor precisión durante los cálculos para reducir los errores de redondeo. Estos formatos extendidos incrementan el tamaño de los exponentes y de las mantisas en una cantidad de bits que puede variar en función de la implementación. Por ejemplo, el formato extendido simple agrega al menos tres bits al exponente y ocho a la mantisa. El formato extendido doble está, en general, formado por 80 bits, con un exponente de 15 bits y 64 bits para la mantisa.

	Valor	Patrón binario		
		Signo	Exponente	Mantisa
(a)	$+1,101 \times 2^5$	0	1000 0100	101 0000 0000 0000 0000 0000
(b)	$-1,01011 \times 2^{-126}$	1	0000 0001	010 1100 0000 0000 0000 0000
(c)	$+1,0 \times 2^{127}$	0	1111 1110	000 0000 0000 0000 0000 0000
(d)	+0	0	0000 0000	000 0000 0000 0000 0000 0000
(e)	-0	1	0000 0000	000 0000 0000 0000 0000 0000
(f)	$+\infty$	0	1111 1111	000 0000 0000 0000 0000 0000
(g)	$+2^{-128}$	0	0000 0000	010 0000 0000 0000 0000 0000
(h)	+NaN	0	1111 1111	011 0111 0000 0000 0000 0000
(i)	$+2^{-128}$	0	011 0111 1111	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

Figura 2.11 • Ejemplos de números en formato de punto flotante, representados según la norma IEEE 754, en formatos de simple precisión (a-h) y de doble precisión (i). Los espacios se usan para facilitar la visualización. No son parte de la representación.

2.3.5.2 Redondeo

Una implementación de la norma IEEE 754 debe proveer al menos un formato de simple precisión, siendo los restantes formatos opcionales. Más aún, el resultado de cualquier operación que se realice con números expresados en formato de punto flotante debe ser precisa en el orden de la mitad del bit menos significativo de la mantisa. Esto significa que durante el cálculo pueden ser necesarios algunos bits adicionales de precisión (**bits de guarda**), y que debe haber un método apropiado para redondear el resultado intermedio al número de bits que constituyen la mantisa.

Existen cuatro formas de redondear números en la norma IEEE 754. Uno de los métodos redondea a cero, el otro redondea a $+\infty$ y el otro, hacia $-\infty$. Por defecto, el redondeo se realiza hacia el número representable más cercano. Los casos intermedios redondean hacia el número cuyo dígito menos significativo es par. Por ejemplo, 1,01101 redondea hacia 1,0110, en tanto que 1,01111 redondea hacia 1,1000.

2.4 Estudio de un caso: una falla en un misil defensivo Patriot causada por una pérdida de precisión

Durante el conflicto bélico ocurrido en los años 1991-1992 entre las fuerzas aliadas y el ejército iraquí, conocida como "Operación tormenta del desierto", las fuerzas aliadas utilizaron una base militar ubicada en Dhahran, Arabia Saudita, la que se hallaba protegida por seis barreras basadas en misiles Patriot de origen estadounidense. El sistema Patriot había sido desarrollado originalmente para ser móvil y para funcionar solo por algunas horas, con el objeto de evitar su detección.

El sistema Patriot rastrea e intercepta ciertos tipos de objetos, como los misiles balísticos Scud, uno de los cuales acertó en una barraca de la marina norteamericana en Dhahran el 5 de febrero de 1991, como consecuencia de lo cual fallecieron 28 norteamericanos. El sistema Patriot falló en el rastreo e intercepción del misil Scud que llegaba, debido a una pérdida de precisión en la conversión de números enteros a formato de punto flotante.

Los sistemas de radar operan a través del envío de un tren de pulsos electromagnéticos y de la posterior escucha de las señales de retorno que pudieran haber sido reflejadas por objetos ubicados en el camino del haz enviado. Si el sistema de radar del Patriot detecta un objeto aéreo de su interés, por ejemplo un Scud, procede a determinar la posición de un detector de rango (véase la figura 2.12), el que procede a estimar la posición que ocupará el objeto rastreado en el momento del próximo barrido. Este detector también permite filtrar la información proveniente de las afueras de los límites establecidos, lo que simplifica el rastreo. La posición del objeto (en este caso un Scud) se valida si se lo encuentra dentro del rango establecido.

La determinación del lugar en que se producirá la próxima aparición del misil depende de la velocidad del mismo. La velocidad del Scud se determina por su cambio de posición con relación al tiempo transcurrido, y el tiempo se actualiza desde el reloj interno del Patriot en intervalos de 100 ms. La velocidad se representa con un número de 24 bits en formato de punto flotante, en tanto que el tiempo se representa como un número entero de 24 bits. Con el objeto de predecir la siguiente aparición del Scud, ambos valores deben representarse como números de 24 bits en formato de punto flotante.

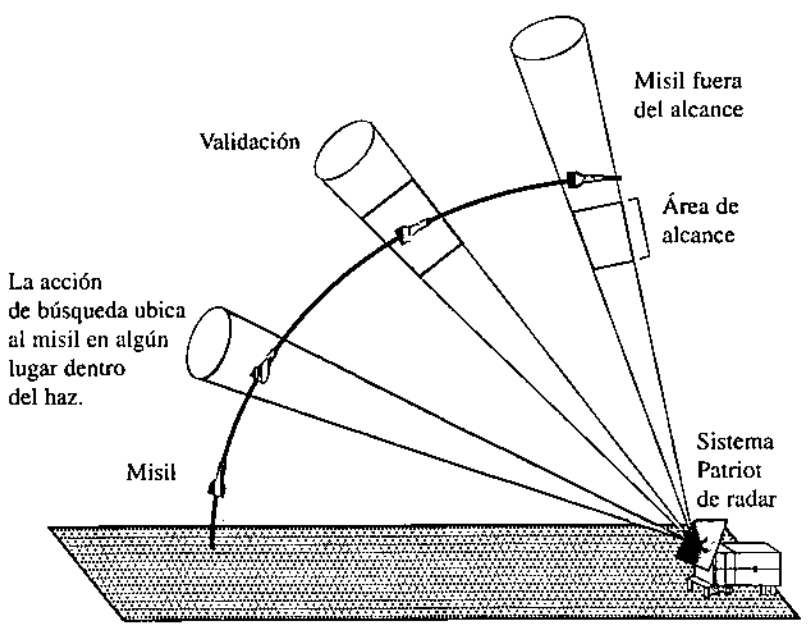


Figura 2.12 • Efecto de los errores de conversión en el cálculo del rango de posición.

La conversión del tiempo desde su formato de punto fijo al formato de punto flotante da por resultado una pérdida de precisión que aumenta en la medida en que aumenta el tiempo medido por el reloj interno. El error introducido por la conversión trae como resultado un error en el cálculo del rango de posición, proporcional a la velocidad del objetivo y del tiempo transcurrido desde el comienzo del funcionamiento del sistema. La causa del incidente de Dhahran, luego de un funcionamiento ininterrumpido del Patriot de más de 100 horas, fue un error de desplazamiento, en el rango calculado, de 687 m, lo que a su vez provocó la fallida intercepción del Scud.

El problema de conversión se conocía desde al menos dos semanas antes del incidente mencionado, por medio de información entregada por Israel; pero la llegada del nuevo software con las correcciones no se produjo sino hasta el día anterior al ataque, debido a la dificultad que implicaba el realizar correcciones del sistema en un ambiente de guerra. Se podría haber adoptado como solución transitoria al problema, hasta tanto se tuviese disponible la modificación requerida de software, la de reiniciar frecuentemente el sistema y, por ende, el reloj. Dado que el personal de campo no tenía información de cuánto tiempo era mucho tiempo en el funcionamiento continuo del sistema, –lo que ya se sabía a partir de la información entregada por Israel–, esta simple solución no se implementó nunca. La lección que deja este caso es que hay que estar muy atento a las limitaciones que implica confiar en operaciones con formatos de precisión finita.

2.5 Códigos alfanuméricos

A diferencia de los números reales, que tienen un rango infinito, existe solo una cantidad finita de caracteres representables. Por consiguiente, para representar un conjunto completo de caracteres hacen falta unos pocos bits por carácter. Se describen aquí tres representaciones habituales en la codificación de caracteres, los códigos alfanuméricos ASCII, EBCDIC y Unicode.

2.5.1 El conjunto de caracteres ASCII

La figura 2.13 representa el conjunto de caracteres que forman el código **ASCII** (*American Standard Code for Information Interchange*), simbolizados en notación hexadecimal. La representación de cada carácter requiere 7 bits, y las 128 combinaciones resultantes son caracteres válidos. Los caracteres representados por los valores hexadecimales 00-1F y el valor 7F corresponden a caracteres especiales de control usados para la transmisión de datos, el control de impresión y otros propósitos no imprimibles. Los caracteres restantes son todos imprimibles, e incluyen la representación de letras, números, símbolos y el símbolo espaciador. Los dígitos 0-9 aparecen en secuencia, tal como lo hacen los dos alfabetos correspondientes a las letras mayús-

culas y minúsculas.[5] Esta organización simplifica el manejo de los caracteres. Para obtener el valor de un dígito a partir de su representación ASCII, se debe restar $(30)_{16}$ de dicha representación. Para convertir el carácter ASCII "5", que se encuentra en la posición $(35)_{16}$, al número 5, se calculará $(35 - 30 = 5)_{16}$. Para convertir una letra mayúscula en su correspondiente minúscula, se le suma $(20)_{16}$ al valor correspondiente a la mayúscula. Por ejemplo, para convertir la letra "H", cuya posición es $(48)_{16}$ en la representación ASCII, en la letra "h", ubicada en la posición $(68)_{16}$, se debe calcular $(48 + 20 = 68)_{16}$.

00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	´	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

NUL	Nulo	FF	Cambio de página	CAN	Cancelación
SOH	Comienzo de encabezado	CR	Retorno a comienzo de línea	EM	Final de medio físico
STX	Comienzo de texto	SO	Shift out	SUB	Sustitución
ETX	Fin de texto	SI	Shift in	ESC	Salida
EOT	Fin de transmisión	DLE	Salida del vínculo de datos	FS	Separaddor de archivos
ENQ	Consulta	DC1	Control de dispositivo 1	GS	Separador de grupos
ACK	Acuse de recibo	DC2	Control de dispositivo 2	RS	Separador de registros
BEL	Campanilla	DC3	Control de dispositivo 3	US	Separador de unidades
BS	Retroceso	DC4	Control de dispositivo 4	SP	Espacio
HT	Tabulación horizontal	NAK	Acuse negativo de recibo	DEL	Borrar
LF	Cambio de línea	SYN	Sincronismo (inactivo)		
VT	Tabulación vertical	ETB	Fin de bloque de transmisión		

Figura 2.13 • El código de caracteres ASCII, representado en valores hexadecimales.

5. Como comentario, los caracteres "a" y "A" son diferentes y tienen códigos diferentes en la representación ASCII. Las letras minúsculas se conocen como de tipo inferior (*lower case*), y las mayúsculas como de tipo superior (*upper case*). La denominación proviene de la posición relativa de los caracteres en los sistemas de tipografía. Las letras mayúsculas aparecen encima de las minúsculas, lo que da origen a la nomenclatura de tipo superior-inferior. En la actualidad, si bien la tipografía se realiza casi siempre en forma electrónica, se sigue manteniendo la nomenclatura tradicional.

2.5.2 El conjunto de caracteres EBCDIC

Un problema que surge en la utilización del código ASCII es su capacidad para representar solo 128 caracteres, lo que plantea una limitación en muchos teclados que además de las letras mayúsculas y minúsculas incluyen una buena cantidad de caracteres especiales.* El código **EBCDIC** (*Extended Binary Coded Decimal Interchange Code*) es un código de ocho bits utilizado ampliamente por las computadoras IBM. Dado que los caracteres ASCII suelen representarse en un formato de ocho bits (un carácter por byte), agregándose a los siete bits ASCII un cero o un uno, el uso de EBCDIC no plantea mayores necesidades de espacio para el almacenamiento de caracteres en una computadora. No obstante, para la transmisión de datos en serie (véase el capítulo 8), el uso de un código de ocho bits requiere tiempos mayores que la transmisión de un código de siete bits, y en este caso, el código de mayor tamaño sí establece una diferencia.

El código EBCDIC se representa en la figura 2.14. Se puede observar la existencia de combinaciones no utilizadas, las que pueden usarse, llegado el caso, para caracteres específicos de alguna aplicación. El hecho de que haya huecos en las secuencias de las letras mayúsculas y minúsculas no genera mayores inconvenientes, dado que el manejo de los caracteres puede seguir haciéndose como en el código ASCII, solo que con diferentes desplazamientos.

2.5.3 El código UNICODE

Los códigos ASCII y EBCDIC sirven para soportar los conjuntos de caracteres (latinos) históricamente dominantes en las representaciones de la computación. Existen muchos otros conjuntos de caracteres en uso en el mundo, y no siempre es posible la conversión de código ASCII al código correspondiente al idioma X. Por consiguiente, se hizo necesario el surgimiento de un nuevo conjunto de caracteres, universal y normalizado, al que se conoce como **Unicode**, y que sirve para soportar una buena cantidad de los alfabetos que se usan en el mundo.

Unicode es una norma en desarrollo, que se modifica a medida que se le incorporan los símbolos correspondientes a alfabetos nuevos, y a medida que los conjuntos de caracteres incorporados se van modificando y se refinan las correspondientes representaciones. En la versión 2.0 de Unicode se incluyen 38.885 caracteres diferentes, los que cubren los principales lenguajes escritos de uso en América, Europa, Medio Oriente, Asia, India y las islas del Pacífico.

---

* *N. de T.*: Los autores se refieren al código ASCII de 7 bits, o estándar, utilizado en el idioma inglés. La limitación que plantean se elimina en el código conocido como ASCII extendido, que con 8 bits admite representar 256 caracteres. De hecho, la traducción de este texto no podría haberse llevado a cabo sin la utilización de los caracteres del lenguaje español incluidos dentro de los 128 caracteres adicionales aportados por el código extendido.

00	NUL	20	DS	40	SP	60	–	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61	/	81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09		29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	'	8A		AA		CA		EA	
0B	VT	2B	CU2	4B		6B	,	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(	6D	_	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F	|	6F	?	8F		AF		CF		EF	
10	DLE	30		50	&	70		90		B0		D0	}	F0	0
11	DC1	31		51		71		91	j	B1		D1	J	F1	1
12	DC2	32	SYN	52		72		92	k	B2		D2	K	F2	2
13	TM	33		53		73		93	l	B3		D3	L	F3	3
14	RES	34	PN	54		74		94	m	B4		D4	M	F4	4
15	NL	35	RS	55		75		95	n	B5		D5	N	F5	5
16	BS	36	UC	56		76		96	o	B6		D6	O	F6	6
17	IL	37	EOT	57		77		97	p	B7		D7	P	F7	7
18	CAN	38		58		78		98	q	B8		D8	Q	F8	8
19	EM	39		59		79		99	r	B9		D9	R	F9	9
1A	CC	3A		5A	!	7A	:	9A		BA		DA		FA	|
1B	CU1	3B	CU3	5B	$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	DC4	5C	*	7C	@	9C		BC		DC		FC	
1D	IGS	3D	NAK	5D	)	7D	'	9D		BD		DD		FD	
1E	IRS	3E		5E	;	7E	=	9E		BE		DE		FE	
1F	IUS	3F	SUB	5F	¬	7F	"	9F		BF		DF		FF	

STX	Comienzo de texto	RS	Separador de registros	DC1	Control de dispostivo1	BEL	Campanilla
DLE	Salida del vínculo de datos	PF	Detener perforador	DC2	Control de dispostivo 2	SP	Espacio
BS	Retroceso	DS	Seleccionar dígito	DC4	Control de dispostivo 4	IL	Inactivo
ACK	Acuse de recibo	PN	Actuar perforado	CU1	Uso del cliente 1	NUL	Nulo
SOH	Comienzo de encabezado	SM	Fijar modo	CU2	Uso del cliente 2		
ENQ	Consulta	LC	Lower Case	CU3	Uso del cliente 3		
ESC	Escape	CC	Cursor Control	SYN	Synchronous Idle		
BYP	Saltear	CR	Acerca de cinta	IFS	Separador de archivos de intercambio		
CAN	Cancelación	EM	End of Medium	EOT	Fin de transmisión		
RES	Reponer	FF	Cambio de página	ETB	Fin del bloque de transmisión		
SI	Shift In	TM	Tape Mark	NAK	Acuse negativo de registro		
SO	Shift Out	UC	Mayúsculas	SMM	Comienzo de mensaje normal		
DEL	Borrar	FS	Separador de archivos	SOS	Co mienzo de significado		
SUB	Reemplazar	HT	Tabulación horizontal	IGS	Separador de grupo de intercambio		
NL	Línea menor	VT	Tabulación vertical	IRS	Separador de registro de intercambio		
LF	Cambio de línea	UC	Mayúsculas	IUS	Separador de unidad de intercambio		

Figura 2.14 • El código alfanumérico EBCDIC, representado con valores hexadecimales.

0000	NUL	0020	SP	0040	@	0060	`	0080	Ctrl	00A0	NBS	00C0	À	00E0	à
0001	SOH	0021	!	0041	A	0061	a	0081	Ctrl	00A1	¡	00C1	Á	00E1	á
0002	STX	0022	"	0042	B	0062	b	0082	Ctrl	00A2	¢	00C2	Â	00E2	â
0003	ETX	0023	#	0043	C	0063	c	0083	Ctrl	00A3	£	00C3	Ã	00E3	ã
0004	EOT	0024	$	0044	D	0064	d	0084	Ctrl	00A4	¤	00C4	Ä	00E4	ä
0005	ENQ	0025	%	0045	E	0065	e	0085	Ctrl	00A5	¥	00C5	Å	00E5	å
0006	ACK	0026	&	0046	F	0066	f	0086	Ctrl	00A6	¦	00C6	Æ	00E6	æ
0007	BEL	0027	'	0047	G	0067	g	0087	Ctrl	00A7	§	00C7	Ç	00E7	ç
0008	BS	0028	(	0048	H	0068	h	0088	Ctrl	00A8	¨	00C8	È	00E8	è
0009	HT	0029	)	0049	I	0069	i	0089	Ctrl	00A9	©	00C9	É	00E9	é
000A	LF	002A	*	004A	J	006A	j	008A	Ctrl	00AA	ª	00CA	Ê	00EA	ê
000B	VT	002B	+	004B	K	006B	k	008B	Ctrl	00AB	«	00CB	Ë	00EB	ë
000C	FF	002C	´	004C	L	006C	l	008C	Ctrl	00AC	¬	00CC	Ì	00EC	ì
000D	CR	002D	-	004D	M	006D	m	008D	Ctrl	00AD	–	00CD	Í	00ED	í
000E	SO	002E	.	004E	N	006E	n	008E	Ctrl	00AE	®	00CE	Î	00EE	î
000F	SI	002F	/	004F	O	006F	o	008F	Ctrl	00AF	¯	00CF	Ï	00EF	ï
0010	DLE	0030	0	0050	P	0070	p	0090	Ctrl	00B0	°	00D0	Đ	00F0	¶
0011	DC1	0031	1	0051	Q	0071	q	0091	Ctrl	00B1	±	00D1	Ñ	00F1	ñ
0012	DC2	0032	2	0052	R	0072	r	0092	Ctrl	00B2	²	00D2	Ò	00F2	ò
0013	DC3	0033	3	0053	S	0073	s	0093	Ctrl	00B3	³	00D3	Ó	00F3	ó
0014	DC4	0034	4	0054	T	0074	t	0094	Ctrl	00B4	´	00D4	Ô	00F4	ô
0015	NAK	0035	5	0055	U	0075	u	0095	Ctrl	00B5	µ	00D5	Õ	00F5	õ
0016	SYN	0036	6	0056	V	0076	v	0096	Ctrl	00B6	¶	00D6	Ö	00F6	ö
0017	ETB	0037	7	0057	W	0077	w	0097	Ctrl	00B7	·	00D7	×	00F7	÷
0018	CAN	0038	8	0058	X	0078	x	0098	Ctrl	00B8	¸	00D8	Ø	00F8	ø
0019	EM	0039	9	0059	Y	0079	y	0099	Ctrl	00B9	¹	00D9	Ù	00F9	ù
001A	SUB	003A	:	005A	Z	007A	z	009A	Ctrl	00BA	º	00DA	Ú	00FA	ú
001B	ESC	003B	;	005B	[	007B	{	009B	Ctrl	00BB	»	00DB	Û	00FB	û
001C	FS	003C	<	005C	\	007C	|	009C	Ctrl	00BC	1/4	00DC	Ü	00FC	ü
001D	GS	003D	=	005D	]	007D	}	009D	Ctrl	00BD	1/2	00DD	Ý	00FD	þ
001E	RS	003E	>	005E	^	007E	~	009E	Ctrl	00BE	3/4	00DE	ý	00FE	þ
001F	US	003F	?	005F	_	007F	DEL	009F	Ctrl	00BF	¿	00DF	ß	00FF	ÿ

NUL	Nulo	SOH	Comienzo de encabezado	CAN	Cancelación	SP	Espacio
STX	Start of text	EOT	Fin de transmisión	EM	Final de medio físico	DEL	Borrar
ETX	End of text	DC1	Control de dispositivo 1	SUB	Sustitución	Ctrl	Control
ENQ	Enquiry	DC2	Control de dispositivo 2	ESC	Escape	FF	Cambio de página
ACK	Acuse de recibo	DC3	Control de dispositivo 3	FS	Separador de campos	CR	Retorno a comienzo de línea
BEL	Campanilla	DC4	Control de dispositivo 4	GS	Separador de grupos	SO	Shift out
BS	Retroceso	NAK	Acuse negativo de recibo	RS	Separador de registros	SI	Shift in
HT	Tabulación horizontal	NBS	Espacio sin interrupción	US	Separador de unidades	DLE	Salida del vínculo de datos
LF	Cambio de línea	ETB	Fin del bloque de transmisión	SYN	Sincronismo (inactivo)	VT	Tabulación vertical

Figura 2.15 • Los primeros 256 códigos de Unicode, con sus representaciones hexadecimales.

La norma Unicode utiliza un conjunto de caracteres de 16 bits, en el que hay correspondencia biunívoca entre los caracteres representados y las palabras de 16 bits. Si bien Unicode soporta muchos más caracteres que ASCII o EBCDIC, no es la norma de mayor importancia. En efecto, la norma Unicode de 16 bits es un subconjunto del Conjunto Universal de Caracteres ISO 10646, de 32 bits de palabra (UCS-4).

Los códigos correspondientes a los 256 primeros caracteres Unicode se muestran en la figura 2.15, de acuerdo con la versión Unicode 2.1. Nótese que los primeros 128 caracteres coinciden con los del ASCII.

Resumen

Toda la información que se maneja dentro de una computadora se representa en términos de bits, los que pueden organizarse e interpretarse como enteros, números de punto fijo, números de punto flotante o caracteres. Los códigos alfanuméricos, tales como ASCII, EBCDIC y Unicode, tienen dimensiones finitas y, por ende, pueden ser representados íntegramente con un número finito de bits. El número de bits que se utiliza para la representación de cantidades también es finito, y como resultado de esta limitación solo puede representarse un subconjunto de la totalidad de los números reales. Esto conduce a las nociones de rango, precisión y error. El rango de una representación numérica define los valores máximo y mínimo que pueden representarse, y queda casi enteramente determinado por la base y la cantidad de bits que se utilizan en el exponente de una representación de punto flotante. La precisión queda determinada por la cantidad de bits usados en la representación de la magnitud, excluyendo los bits correspondientes al exponente en las representaciones de punto flotante. Las representaciones en formato de punto flotante generan errores derivados de la existencia de números que caen dentro de los intervalos existentes entre números consecutivos que pueden ser representados.

Para lectura posterior

D. E. Knuth ofrece un tratamiento completo de los temas de algoritmos y cálculo en computadoras. V. C. Hamacher y otros provee una buena explicación acerca del tema de los errores en las representaciones de punto flotante. La norma de representación de punto flotante IEEE 754 se describe en *IEEE Computer*. Los análisis acerca de rangos, errores y precisión, presentados en la sección 2.3, tuvieron la influencia de G. E. Forsythe. El informe GAO contiene un resumen muy legible del problema de software que provocó la falla del Patriot en Dhahran. Véase http://www.unicode.org para obtener información sobre la norma Unicode.

Forsythe, G. E., "Pitfalls in Computation, or Why a Math Book Isn't Enough", en: *The American Mathematical Monthly*, vol. 77, nº 9, noviembre de 1970, p.p. 931-956.

Hamacher, V. C., Z. G. Vranesic y S. G. Zaky, *Computer Organization*, 3ª ed., McGraw Hill, 1990.

IEEE, "IEEE Standard for Binary Floating Point Arithmetic", ANSI/IEEE Standard 754-1985, en: *IEEE Computer*, vol. 14, marzo de 1981, p.p. 51-62, aparece una edición preliminar.

Knuth, D. E., *The Art of Computer Programming*, vol. *2, Semi-numerical algorithms*, 2ª ed., Addison-Wesley-Longman, 1981.

Informe de la U.S. General Accounting Office GAO/IMTEC-92-26, "Patriot Missile Defense: Software Problem Led to System Failure in Dhahran, Saudi Arabia", U. S. General Accounting Office, P.O. Box 6015, Gaithersburg, Maryland, 20877, febrero de 1992.

Problemas

2.1 Dada una representación de punto fijo, signada, en el sistema decimal, con tres dígitos a la derecha y tres dígitos a la izquierda de la coma decimal, se pregunta:
   a. ¿Cuál es el rango? (Deberán calcularse el máximo número positivo y el mínimo número negativo.)
   b. ¿Cuál es la precisión? (Calcúlese la diferencia entre dos números consecutivos sobre la recta numérica. Recuérdese que el error es la mitad de la precisión.)

2.2 Convertir los números expresados a los sistemas de numeración indicados, utilizando en los resultados la menor cantidad de dígitos.
   a. $(47)_{10}$ a binario sin signo.
   b. $(-27)_{10}$ a binario, magnitud y signo.
   c. $(213)_{16}$ a decimal.
   d. $(10110{,}101)_2$ a decimal.
   e. $(34{,}625)_{10}$ a base 4.

2.3 Convertir los números expresados, a los sistemas de numeración indicados, utilizando en los resultados la menor cantidad de dígitos.
   a. $(011011)_2$ a decimal.
   b. $(-27)_{10}$ a binario exceso 32.
   c. $(011011)_2$ a hexadecimal.
   d. $(55{,}875)_{10}$ a binario sin signo.
   e. $(132{,}2)_4$ a hexadecimal.

2.4 Convertir al sistema decimal el número $(0{,}201)_3$.

2.5 Convertir $(43{,}3)_7$ al sistema octal, usando no más de un dígito octal a la derecha de la coma fraccionaria. Expresar el resultado en una representación octal no signada y truncar los restos por medio de la eliminación de los dígitos sobrantes.

2.6 Representar $(17{,}5)_{10}$ en el sistema de base 3; convertir el resultado obtenido nuevamente al sistema decimal. En la expresión del sistema de base 3 deberán usarse dos dígitos a la derecha de la coma fraccionaria.

2.7 Expresar el equivalente decimal del número binario 1000, considerando que está representado en formato de complemento a dos.

2.8 Expresar el equivalente decimal del número binario 1111, considerando que está representado en formato de complemento a uno.

2.9 Utilizando tres dígitos BCD, representar el número $(305)_{10}$.

2.10 Utilizando tres dígitos BCD, representar el número $(-305)_{10}$ expresado en notación de complemento a 10.

2.11 Para un número dado de bits, indicar si la cantidad de enteros representables en las codificaciones de complemento a uno y complemento a dos son iguales o no.

2.12 Completar la tabla siguiente para las representaciones indicadas, en formato de cinco bits (incluyendo signo). Expresar las respuestas como números enteros decimales signados.

	Magnitud y signo, 5 bits	Exceso 16, cinco bits
Valor máximo		
Valor mínimo (menos negativo		
Cantidad de representaciones		

2.13 Completar la tabla siguiente utilizando notación científica en el sistema binario y una representación de punto flotante de ocho bits, en la que se incluyen un exponente de tres bits expresado en exceso 3 (no exceso 4) y una mantisa normalizada de cuatro bits con un bit implícito. En esta representación, el bit implícito está ubicado a la izquierda de la coma fraccionaria. Esto significa que el número 1,0101 está normalizado, en tanto que 0,101 no lo está.

Notación científica, sistema binario	Representación punto flotante		
	Signo	Exponente	Mantisa
$-1{,}0101 \times 2^{-2}$			
$+1{,}1 \times 2^{2}$			
	0	001	0000
	1	110	1111

2.14 La norma de representación de punto flotante IBM utiliza base 16, un bit de signo, un exponente de siete bits expresado en exceso 64 y una mantisa normalizada de 24 bits.

a. ¿Qué número representa la siguiente combinación binaria?

1 0111111 01110000 00000000 00000000

Expresar la respuesta en el sistema de numeración decimal.

b. Representar en la norma mencionada el número $(14,3)_6$.

2.15 Para una representación de punto flotante normalizada, si se mantienen todos los elementos sin modificar, excepto por:

a. La disminución de la base, se producirá aumento/disminución/ningún cambio en la cantidad de números que pudieran representarse.

b. El incremento de la cantidad de dígitos significativos, se producirá aumento/disminución/ningún cambio en el número positivo más chico que pudiese representarse.
c. El incremento de la cantidad de bits en el exponente, se producirá aumento/disminución/ningún cambio en el rango de representación.
d. El cambio de la representación del exponente a complemento a dos (en vez de exceso 64), se producirá aumento/disminución/ningún cambio en el rango.

2.16 Utilizando una representación de punto flotante con un bit de signo en la posición izquierda del número, seguido por un exponente de dos bits en convención de complemento a dos, y seguido a su vez por una mantisa normalizada expresada en binario, considerando que no hay bit implícito y que el cero se representa como 000000:
a. ¿Cuál es el número decimal representado por la expresión 100100?
b. Si se modifica la base a 4 mientras se mantienen todos los demás elementos constantes, ¿qué ocurre con el mínimo valor que puede representarse?
c. ¿Cuál es el intervalo mínimo entre números consecutivos?
d. ¿Cuál es el intervalo máximo entre números consecutivos?
e. Habiendo un total de seis bits en esta representación, hay un total de 64 combinaciones binarias diferentes. ¿Cuántas de ellas son válidas?

2.17 Representar el número $(107,15)_{10}$ en un formato de punto flotante que ofrezca un bit de signo, un exponente de siete bits en exceso 64 y una mantisa normalizada binaria de 24 bits, sin bit implícito. Truncar la fracción eliminando los bits que fuesen innecesarios.

2.18 Para los valores siguientes, expresados en la norma de punto flotante IEEE 754 de simple precisión, expresar los valores numéricos en la forma de mantisa binaria y exponente (ej.: $1,11 \times 2^5$).
a. 0 10000011 0110000 00000000 00000000
b. 1 10000000 0000000 00000000 00000000
c. 1 00000000 0000000 00000000 00000000
d. 1 11111111 0000000 00000000 00000000
e. 0 11111111 1101000 00000000 00000000
f. 0 00000001 1001000 00000000 00000000
g. 0 00000011 0110100 00000000 00000000

2.19 Indicar los formatos binarios correspondientes a la norma de punto flotante IEEE 754 para los números siguientes.
a. $+1,1011 \times 2^5$ (simple precisión).
b. +0 (simple precisión).
c. $-1,00111 \times 2^{-1}$ (doble precisión).
d. - NaN (simple precisión).

2.20 Utilizando el formato de punto flotante IEEE 754 de simple precisión, indicar el valor (no el patrón binario) correspondiente a:

a. El máximo número positivo representable (infinito no es un número).
b. El mínimo número positivo no nulo normalizado.
c. El mínimo número positivo denormalizado.
d. El mínimo intervalo normalizado.
e. El máximo intervalo normalizado.
f. La cantidad de números normalizados representables (incluyendo el cero; ni el infinito ni el NaN son números).

2.21 Dos programadores escriben generadores de números al azar para números normalizados de punto flotante, usando el mismo método. El programador A utiliza un generador que crea números aleatorios en el intervalo cerrado 0..0,5, y el programador B utiliza uno que crea dichos números en el intervalo cerrado 0,5..1. El generador del programador B funciona correctamente, en tanto que el generador del programador A produce una distribución de números irregular. ¿Cuál puede ser el problema en el planteo del programador A?

2.22 La representación con un bit 1 implícito no funciona en el sistema hexadecimal. ¿Por qué?

2.23 Con una representación que utiliza un bit implícito, ¿puede representarse el valor 0 si todas las combinaciones de bits tanto en el campo de la mantisa como en el del exponente se usan para números no nulos?

2.24 Dado un número en representación de punto flotante expresado en el sistema decimal (por ej.: $0{,}583 \times 10^3$), ¿se puede convertir el número a una forma equivalente en el sistema binario utilizando como método la conversión de la mantisa y del exponente a binario en forma separada?

Capítulo 3

Aritmética

3.1 Introducción

El desarrollo del capítulo anterior permitió explorar algunas de las formas en que puede realizarse la representación de números dentro de una computadora digital, aun cuando el tema de las operaciones aritméticas que pueden ejecutarse sobre dichos números apenas fue tratado. Este capítulo analiza las cuatro operaciones básicas: suma, resta, producto y cociente. Comienza describiendo las formas de operación sobre números de formato de punto fijo y continúa con la descripción de los métodos que permiten la realización de operaciones con números representados en formato de punto flotante.

Algunos de los problemas más importantes, como los cálculos climáticos o las simulaciones de mecánica cuántica, entre otros, ponen a prueba la capacidad de las computadoras existentes hoy en día, aun cuando se trate de las más grandes y poderosas. Por tal razón resulta tan importante el tema del cálculo aritmético de alto rendimiento. El capítulo se completa con la introducción de algunos de los algoritmos y técnicas utilizados para acelerar el proceso de cálculo aritmético.

3.2 Suma y resta en punto fijo

La suma de números binarios y el concepto de desborde (*overflow*) fueron tratados brevemente en el capítulo 2. En este capítulo se tratan en detalle la suma y la resta de números de punto fijo, tanto signados como sin signo. Dado que la representación de enteros en complemento a dos es prácticamente universal en las actuales computadoras, el enfoque se orientará principalmente a las operaciones en complemento a dos. Se analizarán brevemente las operaciones con números representados en complemento a uno y en BCD, que tienen gran significado en otras áreas de la computación tales como la operación en redes (en el caso de sumas en complemento a uno) y en calculadoras portátiles (en el caso de las operaciones en BCD).

3.2.1 Suma y resta en la representación de complemento a dos

En esta sección se analizará la operación de suma de números signados representados en complemento a dos. Durante el análisis de la operación de suma de números signados, la resta de estos quedará implícitamente incluida, como resultado del principio aritmético según el cual:

$$a - b = a + (-b)$$

El correspondiente negativo de un número se puede obtener por medio de su complemento; por consiguiente, se puede realizar una resta por medio de la suma del número complementado. Como consecuencia, se logra un ahorro en la estructura circuital de la unidad de cálculo, dado que se evita la necesidad de un elemento restador por hardware. Este punto se desarrollará en detalle en secciones posteriores.

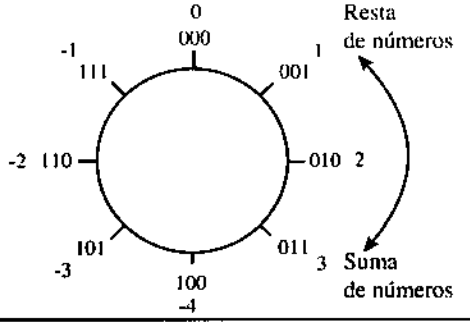


Figura 3.1 • Representación circular de los números de tres bits en representación de complemento a dos.

Lo que sí habrá que hacer cuando se sumen números en representación de complemento a dos es modificar la interpretación de los resultados de la suma. Para entender la razón de esta afirmación, puede considerarse la figura 3.1. Cuando se realizan sumas sobre la recta numérica, los números a sumar pueden ser tan grandes o tan pequeños como se desee; la recta numérica llega hasta $\pm\infty$, por lo que la recta numérica real puede acomodar números de cualquier tamaño. Por otra parte, tal como fuera analizado en el capítulo 2, las computadoras representan la información utilizando un número finito de bits, por lo que solo pueden almacenar valores numéricos dentro de un rango determinado. Por ejemplo, si se analiza la tabla 2.1 se observa que si se restringe el tamaño de un número a, por ejemplo, tres bits, habrá solo ocho posibles combinaciones binarias para representarlo. En la figura 3.1 estos valores se ubican sobre una circunferencia, empezando desde el 000 y avanzando, a lo largo de la misma, hasta llegar a 111 y retornar a 000. La figura muestra también los valores decimales de los mismos números.

Al experimentar con la circunferencia se observa que los números pueden sumarse o restarse recorriendo la circunferencia en el sentido horario para sumar y en el sentido an-

tihorario para restar. También puede restarse por medio de la complementación del sustraendo y posterior suma. Nótese que la situación de desborde solo puede producirse en una suma en la que ambos operandos tienen el mismo signo. Más aún, el desborde se produce cuando ocurre una transición desde +3 a -4 mientras se recorre la circunferencia durante una suma, o desde -4 a +3 durante una resta. (El concepto de desborde en las operaciones de complemento a dos se analiza detalladamente en secciones posteriores del presente capítulo.)

A continuación se plantean dos ejemplos de suma de dos números de ocho bits representados en notación de complemento a dos. En el primero de los dos se plantea la suma de dos números positivos:

	00001010	$(+10)_{10}$
+	00010111	$(+23)_{10}$
	00100001	$(+33)_{10}$

Dos números de signos opuestos se pueden sumar en forma similar:

		00000101	$(+5)_{10}$
	+	11111110	$(-2)_{10}$
Arrastre a descartar	(1)	00000011	$(+3)_{10}$

En las operaciones de suma de números representados en complemento a dos se debe descartar el arrastre producido por la suma en la posición más significativa (el bit a la izquierda). Ocurre una situación similar cuando se genera un arrastre a partir de la suma de dos números negativos:

		11111111	$(-1)_{10}$
	+	11111100	$(-4)_{10}$
Arrastre a descartar	(1)	11111011	$(-5)_{10}$

El bit de arrastre que se produce a la izquierda de la posición más significativa debe descartarse debido a que el sistema numérico es **modular,** es decir, se cierra desde el máximo número positivo sobre el menor número negativo, tal como lo muestra la figura 3.1.*

Si bien la operación de suma puede generar un bit de arrastre desde el bit más significativo del resultado, el descarte del mismo no significa que el resultado sea erróneo. La

* *N. de T.:* Una justificación más rigurosa del motivo por el cual se descarta el bit de arrastre puede encontrarse en la definición del complemento y su aplicación a la resta de números enteros.

sección siguiente analiza en mayor detalle el problema del desborde en la suma de números representados en notación de complemento a dos.

Desborde

Cuando se suman dos números de igual signo, se producirá **desborde** si el resultado es demasiado grande como para poder ser representado con la cantidad de bits utilizados para representar los operandos. Considérese como ejemplo la suma de los números $(+80)_{10}$ y $(+50)_{10}$, expresados ambos en ocho bits. El resultado debería ser $(+130)_{10}$, aunque de la operación representada a continuación se observa que el resultado obtenido es $(-126)_{10}$:

$$\begin{array}{rll} & 01010000 & (+80)_{10} \\ + & 00110010 & (+50)_{10} \\ \hline & 10000010 & (-126)_{10} \end{array}$$

Este resultado no debería sorprender, dado que se sabe que el máximo número positivo que puede representarse en la notación de complemento a dos es $(+127)_{10}$, y que, por lo tanto, es imposible representar $(+130)_{10}$. Si bien el resultado 1000 0010 "se parece" a $(130)_{10}$ si se lo analiza como un número sin signo, al trabajar en forma signada el primer bit, de signo, indica un número negativo, lo que resulta claramente erróneo.

Cuando se suman dos números de signo opuesto no puede ocurrir desborde. Esto se deduce intuitivamente debido a que en una resta la magnitud del resultado no puede superar la magnitud del operando más grande. Esto conduce a la definición de desborde en la representación de números mediante el complemento a dos:

> Si los números que se suman tienen el mismo signo y el resultado tiene el signo opuesto, se ha producido desborde, por lo que el resultado es incorrecto. Si los números que se suman son de signos opuestos, no existe la posibilidad de desborde. Como método alternativo para detectar un desborde en la suma, hay que saber que se produce si y solo si el arrastre que se ingresa hacia el bit de signo difiere del bit de arrastre que sale desde dicho bit.
>
> Si se resta un número positivo de un número negativo y el resultado es positivo, o se resta un número negativo de un número positivo y el resultado es negativo, se produjo desborde. Si los números que se restan son del mismo signo, no se puede producir desborde.

3.2.2 Implementación circuital de sumadores y restadores

Hasta ahora el foco de atención estuvo puesto sobre los algoritmos para la suma y la resta. A continuación se analizan las implementaciones de sumadores y restadores simples.

Sumadores y restadores en serie

El diseño de un **sumador serie** (*ripple carry adder*) para dos números de cuatro bits se analiza en el apéndice A. El circuito sumador sigue el modelo utilizado normalmente para la suma decimal, en el que se suman los dígitos de cada columna, en forma consecutiva una tras otra, de derecha a izquierda. En esta sección se pretende revisar el sumador serie y luego analizar un **restador serie** (*ripple borrow subtractor*), combinando finalmente ambos circuitos para obtener una única unidad de suma y resta.

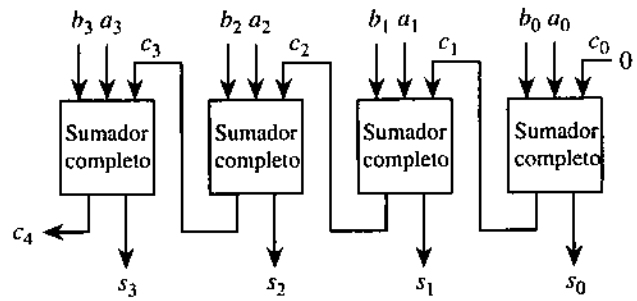


Figura 3.2 • Sumador serie para dos números de cuatro bits.

La figura 3.2 muestra el circuito del sumador serie de cuatro bits desarrollado en el apéndice A. Los dos números binarios *A* y *B* se suman de derecha a izquierda, generando un bit de suma y uno de arrastre en cada columna binaria.

En la figura 3.3 se han conectado en cascada cuatro sumadores binarios serie para permitir la suma de dos números de 16 bits. El sumador completo de la derecha tiene su entrada de arrastre en 0. Si bien este análisis podría haber simplificado el circuito de tal sumador, se prefiere usar la forma más general y forzar c_0 a 0 para simplificar el posterior proceso de resta.

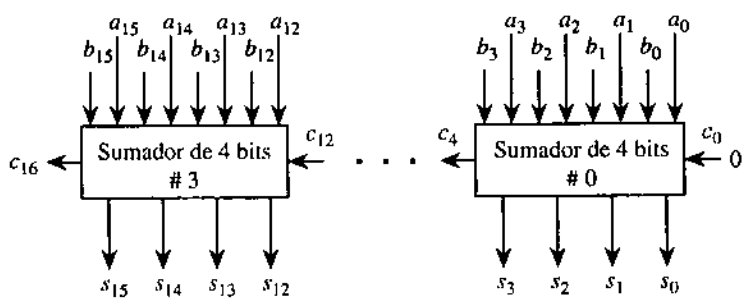


Figura 3.3 • Un sumador de 16 bits obtenido a partir de cuatro sumadores serie de cuatro bits cada uno.

La **resta** de números binarios se resuelve en una forma similar a la de la suma. La resta entre dos números puede realizarse trabajando de a una columna por vez, restando de derecha a izquierda los dígitos del **sustraendo** b_i de los respectivos dígitos del **minuendo** a_i.

Tal como en la resta decimal, si el sustraendo es mayor que el minuendo o hay un arrastre (*borrow*) desde un dígito previo, se deberá propagar dicho arrastre hacia el bit más significativo siguiente. La figura 3.4 ilustra la tabla de verdad y un diagrama en bloque del circuito restador.

a_i	b_i	Arrastre$_i$ (pedido)	Resta$_i$	Arrastre$_{i+1}$ (pedido)
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

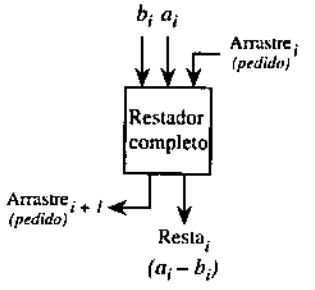


Figura 3.4 • Tabla de verdad y símbolo esquemático para un restador serie de un bit.

Los circuitos restadores de dos números de un bit pueden conectarse en cascada para crear un circuito **restador serie**, de la misma manera en que se conectan en serie los sumadores completos de números de un bit para formar un sumador serie. La figura 3.5 ilustra un circuito restador serie de cuatro bits formado por cuatro restadores completos de un bit.

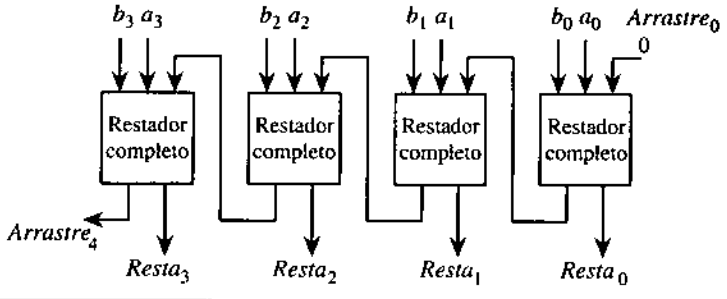


Figura 3.5 • Restador serie.

Tal como se analizara previamente, un método alternativo para la implementación de la resta consiste en determinar el complemento a dos del sustraendo y sumarlo al valor del minuendo. El circuito de la figura 3.6 resuelve tanto la operación de suma como la de resta sobre dos números de cuatro bits, para lo cual requiere que las entradas b_i puedan complementarse cuando se desea realizar una resta. Una línea de control $\overline{\text{SUMAR}}$/RESTAR determina cuál es la operación a realizar. La barra por encima del símbolo SUMAR indica que la operación de suma se activa cuando la señal está en su nivel bajo. Esto es, si la línea de control está en su estado 0, las entradas a_i y b_i se transfieren al sumador, el que genera la suma s_i en sus salidas. Si la línea de control está en 1, las entradas a_i se transfie-

ren al sumador, pero las entradas b_i se complementan a uno en las compuertas XOR antes de su ingreso al sumador. Para formar el complemento a dos, se debe sumar 1 al complemento a 1, lo que se logra colocando la línea de arrastre de entrada c_0 en 1 por medio de la entrada de control. De esta manera, la electrónica del sumador se comparte entre el sumador y el restador.

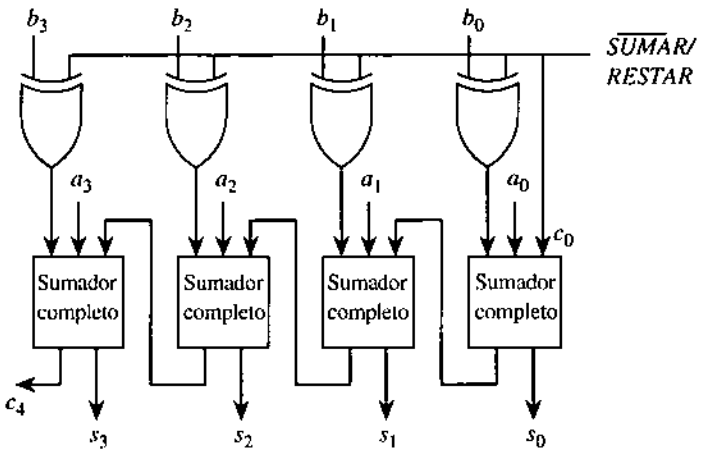


Figura 3.6 • Circuito sumador-restador.

3.2.3 Suma y resta en representación de complemento a uno

Si bien la representación de complemento a uno actualmente no se utiliza demasiado en los sistemas de cómputo, fue muy usada en las primeras computadoras. La suma en complemento a uno se maneja en forma un poco diferente a la suma en la representación de complemento a dos: el bit de arrastre que se genera a partir de la posición más significativa no se descarta sino que se vuelve a sumar con la posición menos significativa del resultado obtenido, tal como lo muestra la figura 3.7. Esta operación suele conocerse como **arrastre circular final**.

Para entender mejor la causa por la cual se necesita reutilizar el arrastre, puede examinarse la circunferencia mediante la que se representan los números de complemento a uno, la que se muestra en la figura 3.8. Nótese que la circunferencia ofrece dos representaciones para el 0. Cuando se suman dos números, al atravesar tanto una como otra representación del cero, debe compensarse el hecho de que el 0 se recorre dos veces. El agregado del bit de arrastre a la columna de las unidades desplaza en uno el resultado final para salvar esta situación.

Debe notarse aquí que la distancia entre 0 y -0 es la distancia entre dos enteros y no la distancia entre dos números representables consecutivos. Como aclaración de este punto, considérese la suma de $(+5,5)_{10}$ con $(-1,0)_{10}$, representados ambos en la notación de complemento a uno, según la figura 3.9. (Cabe aclarar que este problema podría tratarse tam-

```
  10011   (-12)10
+ 01101   (+13)10
-------
 100000
   └──┐   Corrección del resultado
      ▼
+       1
-------
  00001   (+1)10
```

Figura 3.7 • Ejemplo de la suma en complemento a uno con reutilización del arrastre generado.

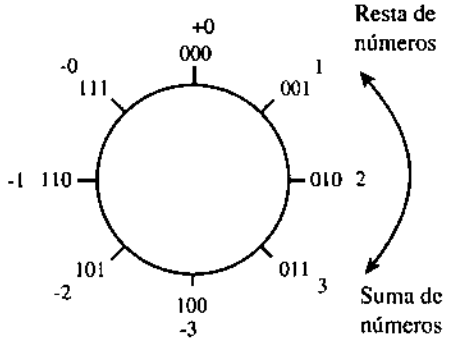


Figura 3.8 • Circunferencia representativa de los números en notación de complemento a uno.

bién como una resta, en la cual se complementa el sustraendo, por inversión de todos sus bits, antes de sumarlo al minuendo). Para poder sumar $(+5,5)_{10}$ y $(-1,0)_{10}$ y obtener el resultado correcto en la representación de complemento a uno, se suma a la posición de las unidades el arrastre generado por la suma realizada, según se muestra. Esto complica la representación circular de los números, dado que en el intervalo entre +0 y -0 existen números válidos que representan fracciones negativas, pero que aparecen en la circunferencia antes de que aparezca la representación del número -0. Si se reordena la circunferencia de modo de salvar esta anomalía, habrá que tratar la suma en una forma más complicada.

La aparición de dos representaciones diferentes para el cero y la necesidad potencial de realizar otra suma para agregar el bit de arrastre son dos razones importantes para que los diseñadores prefieran la aritmética de complemento a dos antes que la de complemento a uno.

```
   0101,1   (+5,5)10
+  1110,0   (-1,0)10
-------
  10011,1
+ └──→1,0
-------
   0100,1   (+4,5)10
```

Figura 3.9 • La suma del bit de arrastre complica la operación de suma para números no enteros.

3.3 Producto y cociente en punto fijo

La multiplicación y la división de números representados en punto fijo pueden resolverse mediante operaciones más sencillas de suma, resta y desplazamiento. Las secciones que siguen describen métodos para realizar las operaciones de producto y cociente de números en representación de punto fijo, tanto signados como no signados, utilizando las operaciones básicas mencionadas. Se analizarán primeramente las operaciones para números no signados y, luego, las operaciones para números signados.

3.3.1 Multiplicación de números sin signo

La multiplicación de números binarios enteros sin signo se realiza en la misma forma en que se la realiza manualmente para los números decimales. La figura 3.10 ilustra el procedimiento para multiplicar dos enteros binarios sin signo. Cada bit del multiplicador determina si el multiplicando, desplazado a la izquierda de acuerdo con la posición del bit del multiplicador, se suma o no al producto. Cuando se multiplican dos números sin signo de n bits, el resultado puede estar formado por hasta $2n$ bits. Para el ejemplo de la figura 3.10, el producto de dos operandos de cuatro bits cada uno da por resultado un valor de ocho bits. Cuando se multiplican dos números signados de n bits, el resultado solo puede estar formado por $2(n-1)+1=(2n-1)$ bits, siendo este valor el equivalente a realizar el producto de dos números sin signo de $(n-1)$ bits cada uno y agregar luego el bit de signo.

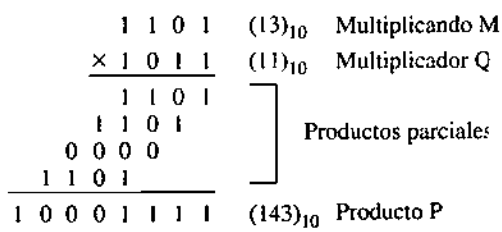


Figura 3.10 • Multiplicación de dos números enteros binarios sin signo.

La implementación circuital del producto de enteros puede adoptar una forma similar a la que se utiliza en el método manual. La figura 3.11 ilustra el circuito de una unidad multiplicadora para números de cuatro bits, la que contiene un sumador de cuatro bits, una unidad de control, tres registros de cuatro bits y un registro de un bit para el arrastre. Para realizar el producto de dos números, el multiplicando se coloca en el registro M, el multiplicador en el registro Q, en tanto que los registros A y C se colocan en cero. Durante la operación, el bit de la derecha del multiplicador determina, en cada paso, si el multiplicando debe sumarse o no al producto parcial. Luego de sumar el multiplicando al producto, el multiplicador y el registro A se desplazan simultáneamente a derecha. Esto produce el efecto de desplazar el multiplicando a la izquierda (tal como se realiza

en el procedimiento manual) y, a la vez, de colocar el siguiente bit del multiplicador en la posición q_0.*

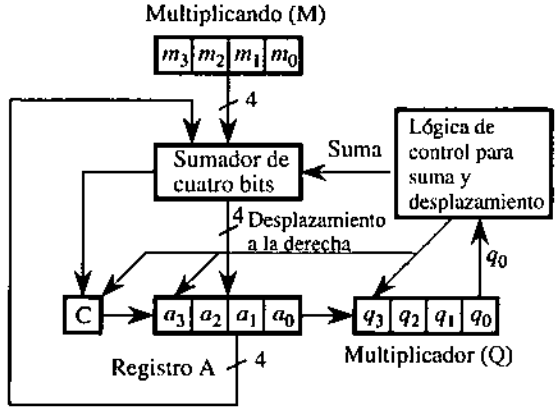


Figura 3.11 • El multiplicador serie.

La figura 3.12 muestra el proceso de multiplicación. Al comenzar, C y A se limpian, en tanto que M y Q contienen, respectivamente, al multiplicando y al multiplicador. El bit menos significativo de Q es 1, y, por consiguiente, el multiplicador M se suma con el producto contenido en el registro A. La palabra formada por los contenidos de los registros A y Q tomados como un conjunto constituye el producto de ocho bits, siendo el registro A el que recibe la suma del multiplicando. Luego de sumar M con A, los registros A y Q se desplazan a derecha. Dado que los registros A y Q se vinculan como un par para contener el producto de ocho bits, el bit menos significativo de A se desplaza hacia la posición más significativa de Q. El bit menos significativo de Q se descarta, C se desplaza hacia la posición más significativa de A y se ingresa un 0 en C.

El proceso continúa durante tantos pasos como bits tenga el multiplicador. En la segunda iteración, el bit menos significativo de Q vuelve a ser 1, por lo que el multiplicando se suma con A, desplazándose la combinación de los registros C, A y Q hacia la derecha. En la tercera iteración, el bit menos significativo de Q es 0, por lo que M no se suma con A, pero sí se produce el desplazamiento a derecha de los tres registros. Finalmente, en la cuarta iteración, el bit menos significativo de Q vuelve a ser 1, por lo que nuevamente M se suma con A y se desplaza a derecha la combinación de los tres registros C, A y Q. El producto queda contenido en los registros A y Q, donde el registro A contiene los bits más significativos, en tanto que Q contiene los bits de menor peso.

* *N. de T.*: Juntamente con el desplazamiento de los dos registros mencionados, se procede a la carga del bit de arrastre en el registro C.

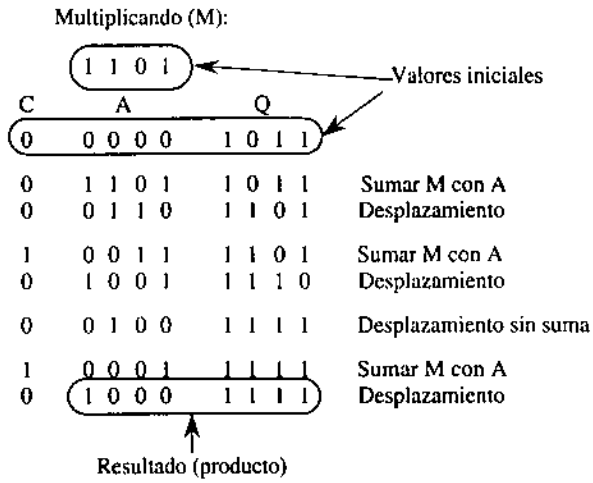


Figura 3.12 • Un ejemplo de producto utilizando el multiplicador serie.

3.3.2 División sin signo

En la división binaria, se debe intentar reiteradamente la resta del dividendo menos el divisor, usando la menor cantidad de bits posibles en el dividendo. La figura 3.13 ilustra este punto mostrando que $(11)_2$ no cabe en 0 o en 01, pero sí cabe en 011, como se puede ver desde el patrón binario 001 que inicia el cociente.

```
0111 | 11
  11   0010
  --
  01 <-- Resto
```

Figura 3.13 • Ejemplo de división binaria.

La división de números enteros binarios en una computadora se puede manejar en forma similar a la que se utiliza para el producto de enteros binarios, pero con la dificultad de tener que determinar si el dividendo cabe o no en el divisor, cuya única solución es efectuar la resta y verificar si el resto es negativo. Si eso ocurre, se debe dar marcha atrás con la resta, para lo cual se vuelve a sumar el divisor, como se describe más adelante.

En el algoritmo de división, en lugar de desplazar el producto a la derecha tal como se realizara en el caso del producto, se desplaza el cociente a la derecha y se resta en vez de sumar. Cuando se dividen dos números sin signo formados por n bits, el resultado no puede tener más que n bits.

La figura 3.14 ilustra el circuito de una unidad divisora para números de cuatro bits, la que contiene un sumador de cinco bits, una unidad de control, un registro de cuatro bits para el dividendo Q, y dos registros de cinco bits para el divisor M y para el resto A. Se utilizan registros de cinco bits para A y M, en lugar de usar registros de 4 bits como po-

dría suponerse, dado que se requiere un bit adicional para indicar el signo de los resultados intermedios. Si bien este método de división se usa para números sin signo, se utilizan restas a lo largo del proceso, por lo que pueden obtenerse resultados negativos. Esto extiende el rango de -16 a +15, lo cual, a su vez, plantea la necesidad de 5 bits para el almacenamiento de los resultados intermedios.

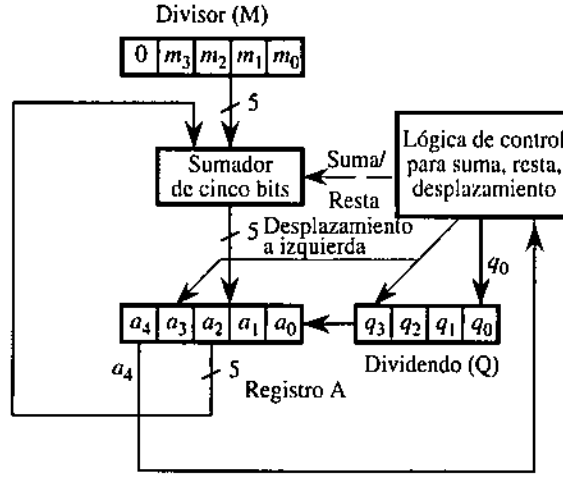


Figura 3.14 • Circuito divisor serie.

Para dividir dos números de cuatro bits, se almacena el dividendo en el registro Q y el divisor en el registro M, en tanto que el registro A y el bit más significativo de M se cargan con 0. El bit más significativo del registro A determina en cada paso si el divisor debe volver a sumarse al dividendo. Esto se hace necesario para reponer el dividendo cuando, como se ha descripto anteriormente, el resultado de restar el divisor resulta negativo. Se suele hablar de una **división con reposición** porque, en los casos en que el resto es negativo, se debe restaurar el dividendo a su valor anterior. Cuando el resultado no es negativo, el bit menos significativo de Q se lleva a 1 para indicar que el divisor cabe en el dividendo.

En la figura 3.15 se ilustra el proceso de división. El registro A y el bit más significativo de M se llevan a cero, en tanto que Q y los bits menos significativos de M se cargan con el dividendo y con el divisor, respectivamente. Los registros A y Q se desplazan en conjunto a la izquierda y se resta el divisor M de A. Dado que el resultado es negativo, el divisor se vuelve a sumar para reponer el dividendo, y q_0 se carga con 0. El proceso se repite, desplazando A y Q a izquierda y restando M de A. Nuevamente se obtiene un resultado negativo, por lo que se vuelve a reponer el dividendo y se coloca un cero en q_0. En la tercera iteración, se desplazan a izquierda A y Q, en tanto M se vuelve a restar de A. En este caso, el resultado de la resta no es negativo, por lo que se coloca un 1 en q_0. El proceso se repite una vez más, para realizar una iteración final, en la que nuevamente A y Q se desplazan a izquierda y M se resta de A, lo que vuelve a dar un resultado negativo. Se

repone el dividendo y se carga un 0 en q_0. El cociente queda contenido en el registro Q, en tanto que el registro A contiene el resto de la división.

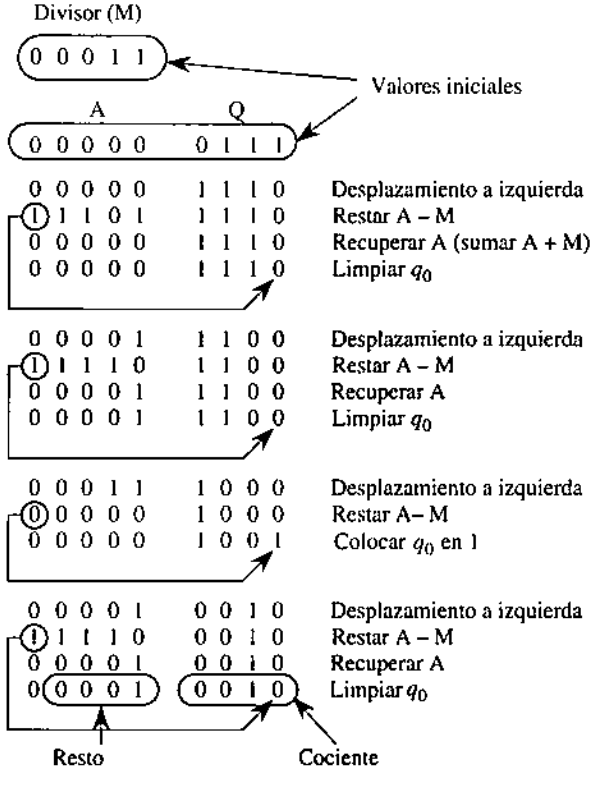


Figura 3.15 • Un ejemplo de división que utiliza el divisor serie.

3.3.3 Producto y cociente signados

Si se aplican los métodos descriptos en la sección anterior al producto y cociente de números con signo, surgirán algunos problemas. Considérese el producto de -1 por +1 utilizando palabras de cuatro bits, como se muestra en el esquema de la izquierda de la figura 3.16. En lugar de obtener el resultado esperado de -1, se obtiene el equivalente a +15. Lo ocurrido se debe a que el bit de signo no se extendió hacia la izquierda del resultado. Este no es un problema cuando el resultado es positivo, dado que los bits de mayor peso adoptan el valor 0, por lo que el bit de signo se genera correctamente como 0.

En el esquema de la derecha de la figura 3.16 se puede observar una solución, en la cual cada uno de los productos parciales se extiende a todo lo ancho de la palabra resultado, reteniéndose solo los bits menos significativos de dicho resultado. Si ambos operandos son negativos, se extienden ambos signos, nuevamente reteniendo solo los ocho bits menos significativos del resultado.

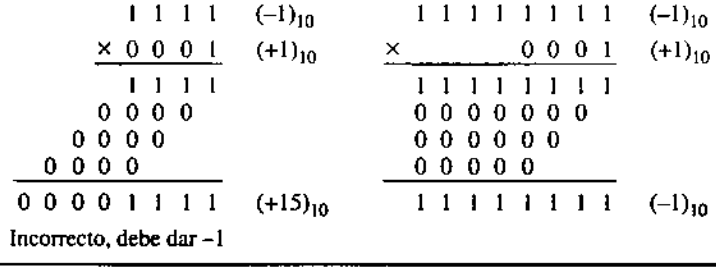


Figura 3.16 • Producto de números con signo.

La división signada es más compleja. No es intención de este texto el desarrollo de los métodos requeridos, pero, como técnica general, la solución consiste en convertir ambos operandos a su forma positiva, realizar la operación y luego, como paso final, convertir el resultado a su forma signada correcta.

3.4 Aritmética de punto flotante

Las operaciones aritméticas que utilizan números representados en notación de punto flotante pueden realizarse mediante las operaciones de punto fijo descriptas en las secciones anteriores, prestando la debida atención a la necesidad de mantener las características fundamentales de la representación de punto flotante. En las secciones siguientes se explorarán los procedimientos para la realización de cálculos tanto en el sistema de numeración binario como en el decimal, sin olvidar los requisitos propios de la representación de punto flotante.

3.4.1 Suma y resta en formato de punto flotante

Las operaciones aritméticas de punto flotante difieren de las de punto fijo en el hecho de que, además de considerarse las magnitudes de los operandos, tiene que considerarse también el tratamiento que debe darse a sus exponentes. Como en el caso habitual de las operaciones decimales en notación científica, los exponentes de los operandos deben ser iguales para poder sumar o restar. Se suman o restan las mantisas según corresponda, y se completa la operación normalizando el resultado.

Los procesos de ajuste de la parte fraccionaria y de redondeo del resultado pueden llevar a una pérdida de precisión del resultado. Como ejemplo, considérese la suma sin signo de los operandos $(0{,}101 \times 2^3) + (0{,}111 \times 2^4)$, en los que las mantisas se representan con tres dígitos significativos. Se inicia la operación haciendo que el menor de los exponentes se iguale al mayor de ellos, y se ajusta adecuadamente la mantisa. Se tendrá así que $0{,}101 \times 2^3 = 0{,}010 \times 2^4$, por lo que se pierde en este proceso una precisión equivalente a $0{,}001 \times 2^3$. La suma resultante es:

$$(0{,}010 + 0{,}111) \times 2^4 = 1{,}001 \times 2^4 = 0{,}1001 \times 2^5$$

y si se vuelve a redondear a tres dígitos significativos, se obtiene $0{,}100 \times 2^5$, lo que implica otra pérdida de precisión de $0{,}001 \times 2^4$ en el proceso de redondeo.

¿Por qué los números de punto flotante tienen formatos tan complicados?

Puede ser motivo de curiosidad saber la causa de la complejidad de la estructura que se utiliza cuando se representan números en formato de punto flotante. La mantisa se representa en formato de magnitud y signo; el exponente, a su vez, en notación excedida, y el bit de signo separado del resto de la mantisa por el campo de exponente. La explicación de esta estructura es relativamente simple. Considérese la complejidad con la que se realizan las operaciones de punto flotante en una computadora. Antes de poder realizar cálculo alguno, el número debe ser desempaquetado a partir del formato utilizado para su almacenamiento. (En el capítulo 2 se analiza el formato de representación de números en punto flotante descripto en la norma IEEE 754.) Antes de realizar la operación, se deben extraer del conjunto de bits la mantisa y el exponente; luego de realizar la operación aritmética, el resultado debe volver a normalizarse y redondearse, y finalmente se deben reacomodar los bits para satisfacer los formatos requeridos.

La ventaja del formato de punto flotante que contiene un bit de signo seguido por un exponente en notación excedida, seguido a su vez por la magnitud de la mantisa, es que permite la realización de comparaciones entre dos números de punto flotante, por mayor, menor o igual, sin necesidad de desempaquetar los números. El bit de signo es lo más importante en tal comparación, por lo que es razonable que el signo, en el formato de punto flotante, se encuentre en el bit más significativo. Lo que sigue en importancia, en la comparación, es el exponente, dado que una variación de ±1 en el exponente modifica el valor en un factor de 2 en el sistema binario, en tanto que un cambio en cualquier bit de la mantisa, aun en el más significativo, modifica el número en un valor menor.

Para tener en cuenta el bit de signo, las mantisas signadas se representan como enteros y se convierten en su formato de complemento a dos. Una vez realizada la suma o la resta en formato de complemento a dos, puede haber necesidad de normalizar el resultado y de ajustar el bit de signo, tras lo cual nuevamente se lleva el resultado al formato de representación de magnitud y signo.

3.4.2 Producto y cociente en formato de punto flotante

El producto y el cociente en el formato de punto flotante se resuelven en forma similar a la de la suma y la resta en punto flotante, excepto por el hecho de que tanto el signo como el exponente y la mantisa del resultado se pueden calcular por separado. Si los operandos tienen el mismo signo, el signo del resultado es positivo. Signos distintos produ-

cen un resultado negativo. El exponente del resultado, antes de su normalización, se obtiene sumando los exponentes de los factores en la multiplicación, o restando los exponentes de dividendo y divisor en la división. Las mantisas se multiplican o dividen de acuerdo con la operación a realizar, tras lo cual se normaliza el resultado.

Considérese el uso de mantisas de tres bits al calcular el producto binario de los números $(+0,101 \times 2^2) \times (-0,110 \times 2^{-3})$. Los signos de los operandos difieren, lo que significa que el resultado tendrá signo negativo. Para multiplicar se suman los exponentes, por lo que el exponente del resultado será $2 + (-3) = -1$. Se multiplican las mantisas, de donde se obtiene el producto 0,01111. La normalización del producto y su redondeo a solo tres bits en la mantisa produce como resultado el valor $-0,111 \times 2^{-2}$.

Ahora se considera la división de los números binarios $(+0,110 \times 2^5)$ y $(+0,100 \times 2^4)$, cuyas mantisas se representan con tres bits. Los signos de los operandos son iguales, por ende, el resultado tiene signo positivo. Para dividir se restan los exponentes, por lo que el exponente del resultado es $5 - 4 = 1$. Se dividen luego las mantisas, lo que puede realizarse en distintas formas. Si se tratan las mantisas como enteros no signados, se obtiene 110/100 = 1 con un resto de 10. Lo que se pretende en realidad es un conjunto de bits que representen la mantisa en vez de dos elementos que en forma separada representen cociente y resto. Para lograrlo, se puede escalar el dividendo en dos posiciones a la izquierda, con lo que se obtiene el resultado 11000/100 = 110. Si el resultado obtenido se vuelve a escalar dos lugares, ahora a la derecha, para mantener el factor de escala original, se logra 1,1. Si se agrupan todos los elementos obtenidos, el resultado de dividir $(+0,110 \times 2^5)$ por $(0,100 \times 2^4)$ es $(+1,10 \times 2^1)$, el que, luego de la normalización, se convierte en $(+0,110 \times 2^2)$.

3.5 Aritmética de alto rendimiento

La velocidad con la que se realizan las operaciones aritméticas suele ser el cuello de botella en el rendimiento de una computadora. Muchas supercomputadoras, como la Cray, la Tera y la Intel Hipercubo, se consideran "super" debido a su eficiencia en el cálculo tanto de punto fijo como flotante. En esta sección, se analizan distintas técnicas para mejorar la velocidad de las operaciones de suma, resta, producto y cociente.

3.5.1 Suma de alto rendimiento

El sumador serie analizado en la sección 3.2.2 puede introducir retardos importantes en el sistema que lo utilice. El camino más largo dentro del sumador es aquel que va desde las entradas del sumador menos significativo hasta las salidas del sumador más significativo. El proceso de sumar las entradas correspondientes a cada posición binaria es relativamente rápido (es suficiente un circuito en lógica de dos niveles), pero la propagación del arrastre demora un tiempo largo en recorrer el circuito. En efecto, el tiempo de propagación es proporcional a la cantidad de bits de los operandos. La consecuencia es

desafortunada, dado que aumentar la cantidad de dígitos significativos en una suma se traduce en un mayor requerimiento de tiempo para realizar la suma. En esta sección se analiza un método para acelerar la propagación del arrastre, en lo que se conoce como un **sumador con arrastre anticipado** (*carry lookahead adder*).

En el apéndice B se desarrollan las expresiones booleanas reducidas correspondientes a la suma (s_i) y el arrastre (c_{i+1}) de un sumador completo. Se reiteran aquí estas expresiones, donde los subíndices indican la posición relativa de cada sumador completo de un bit en el sumador serie:

$$s_i = \bar{a}_i\bar{b}_i c_i + \bar{a}_i b_i \bar{c}_i + a_i\bar{b}_i\bar{c}_i + a_i b_i c_i$$

$$c_{i+1} = b_i c_i + a_i c_i + a_i b_i$$

Si se factorea la segunda ecuación se obtiene

$$c_{i+1} = a_i b_i + (a_i + b_i)c_i$$

La que a su vez puede escribirse en la forma

$$c_{i+1} = G_i + P_i c_i$$

En la que

$$G_i = a_i b_i \text{ y } P_i = a_i + b_i$$

Los términos G_i y P_i se conocen como funciones de **generación** y **propagación**, respectivamente, por el efecto que tienen sobre el arrastre. Cuando $G_i = 1$, se genera un arrastre en la etapa i. Cuando $P_i = 1$, se propaga un arrastre a través de la etapa i siempre que alguno de los operandos a_i o b_i valgan 1. Los términos G_i y P_i pueden implementarse con lógica de único nivel ya que solo dependen, respectivamente, de un producto lógico (Y) o de una suma lógica (O) de las variables de entrada.

Nuevamente, los arrastres requieren el mayor tiempo. La salida de la etapa 0 correspondiente al arrastre c_1 vale $G_0 + P_0 \cdot c_0$, y dado que en la suma $c_0 = 0$, esta expresión puede escribirse como $c_1 = G_0$. La salida de la etapa 1 correspondiente a c_2 es $G_1 + P_1 \cdot c_1$, y dado que $c_1 = G_0$, esta expresión se escribe también como $c_2 = G_1 + P_1 \cdot G_0$. La salida de arrastre c_3 de la etapa 2 vale $c_3 = G_2 + P_2 \cdot c_2$, y dado que $c_2 = G_1 + P_1 \cdot G_0$, también puede escribirse la expresión como $c_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0$. Reiterando este planteo por última vez para un sumador de cuatro bits, la salida de arrastre que se obtiene desde la etapa 3 es $c_4 = G_3 + P_3 \cdot c_3$, por lo que, considerando el valor obtenido para c_3, se puede escribir esta última expresión como

$$c_4 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0$$

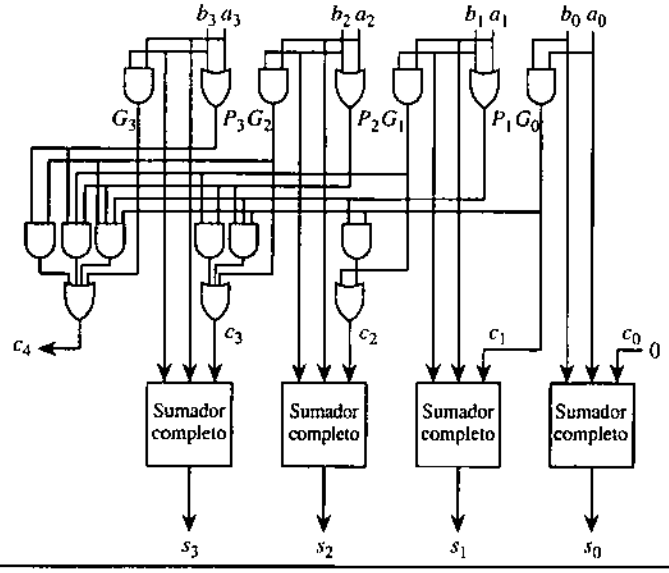


Figura 3.17 • Sumador con arrastre anticipado.

Ahora puede implementarse un sumador de cuatro bits con arrastre anticipado, tal como el de la figura 3.17. Sigue existiendo el retardo entre los sumadores al igual que en el caso anterior, pero ahora la cadena de arrastres puede pensarse como un conjunto de partes independientes que requieren el retardo de una compuerta para G_i y P_i, y dos retardos adicionales para generar c_{i+1}. Por lo tanto, se agrega un retardo equivalente al tiempo de propagación de tres compuertas, pero se elimina la cadena de arrastres entre las distintas etapas. Si se supone que cada sumador completo de un bit introduce un retardo equivalente a dos compuertas, un sumador de cuatro bits con arrastre anticipado tendrá un retardo máximo equivalente a cinco compuertas, en tanto que en el caso de un sumador serie de cuatro bits el retardo máximo será equivalente al tiempo de propagación de ocho compuertas. La diferencia entre las dos soluciones se vuelve más pronunciada cuanto más grandes son los operandos. Este procedimiento suele estar acotado a alrededor de ocho bits de anticipación, debido a las limitaciones en los factores de carga de las compuertas analizadas en el apéndice A. Para sumar números de más de ocho bits, es posible plantear circuitos parciales de anticipo de arrastre, los que pueden encadenarse para calcular las entradas y salidas de arrastre de cada una de esas etapas. (Véase el ejemplo de la página 86.)

3.5.2 Producto de alto rendimiento

Existen varios métodos para acelerar el proceso de la multiplicación. Dos de estos métodos se describen en las secciones siguientes. El primero gana en eficiencia mediante el esquema de saltear bloques de unos, lo que elimina pasos de suma en la opera-

ción. Luego, se describe un multiplicador paralelo, en el que se forma un producto cruzado de todos los posibles pares de bits del multiplicando y del multiplicador. El producto final se obtiene cuando se suma por filas el resultado del producto cruzado antes generado.

El algoritmo de Booth

El algoritmo de Booth trata a los números positivos o negativos en la misma forma. Su funcionamiento se basa en el hecho de que, cuando en el multiplicador existen secuencias de ceros o unos, no se requieren sumas, sino solo desplazamientos. Las sumas o las restas se llevan a cabo en los límites de las secuencias, donde se detecten transiciones de 0 a 1 o de 1 a 0. Una secuencia de unos en el multiplicador, ubicada entre las posiciones de pesos 2^u a 2^v puede considerarse como $2^{u+1} - 2^v$. Por ejemplo, si el multiplicador es 001110 $(+14)_{10}$, $u = 3$ y $v = 1$, por lo que $2^4 - 2^1 = 14$.

En la implementación circuital se analiza el multiplicador de derecha a izquierda. La primera transición que se detecta es un cambio de 0 a 1, lo que requiere la resta del valor inicial (0) menos 2^1. En la transición siguiente, de 1 a 0, se suma 2^4, lo que da por resultado +14. Se debe considerar que existe un 0 agregado a la derecha del multiplicador con el objeto de definir la situación en la que aparezca un 1 como dígito menos significativo del mismo.

Si el multiplicador se codifica de acuerdo con el algoritmo de Booth, el proceso de la multiplicación puede llegar a requerir menos pasos. Considérese el ejemplo de multiplicación ilustrado por la figura 3.18. El multiplicador $(14)_{10}$ contiene tres unos consecutivos, lo que implica, si se usara el procedimiento de multiplicar por sumas y desplazamientos descripto en la sección 3.3.1, la necesidad de tres operaciones de suma. El multiplicador codificado según Booth se obtiene analizando el multiplicador original de derecha a izquierda, colocando un -1 en la primera posición en la que se encuentre un 1, y colocando un +1 en la posición en la que aparezca el siguiente 0. El multiplicador se convierte así en 0 +100 -1 0. El multiplicador codificado según Booth solo contiene dos posiciones no nulas, lo que significa que habrá que realizar una única suma y una única resta, por lo que este ejemplo logra una reducción en el tiempo requerido para completar el producto.

Sin embargo, no siempre se logran mejoras, y en algunos casos puede darse que el algoritmo en cuestión genere la realización de más operaciones que si no se lo hubiese utilizado. Al efecto, considérese el ejemplo de la figura 3.19, en la que el multiplicador consiste en una secuencia de ceros y unos alternados. Este ejemplo es el mismo de la figura 3.18, en la cual se han conmutado multiplicando y multiplicador. Sin la codificación Booth del multiplicador se hubiesen requerido tres operaciones de suma dado que tiene tres unos. Si se lo codifica según Booth, se requerirán seis operaciones de suma y resta, lo que resulta notablemente peor. La sección siguiente intenta mejorar esta situación.

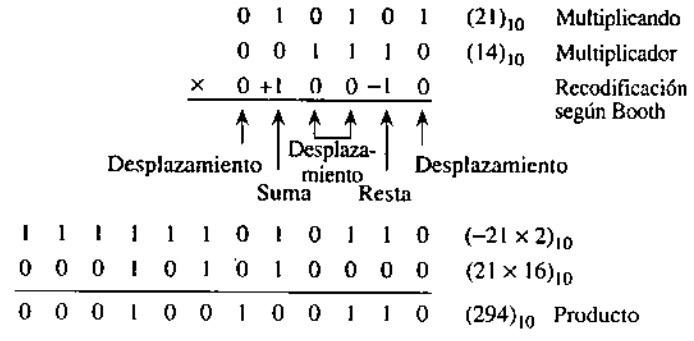


Figura 3.18 • Producto de enteros con signo mediante el algoritmo de Booth.

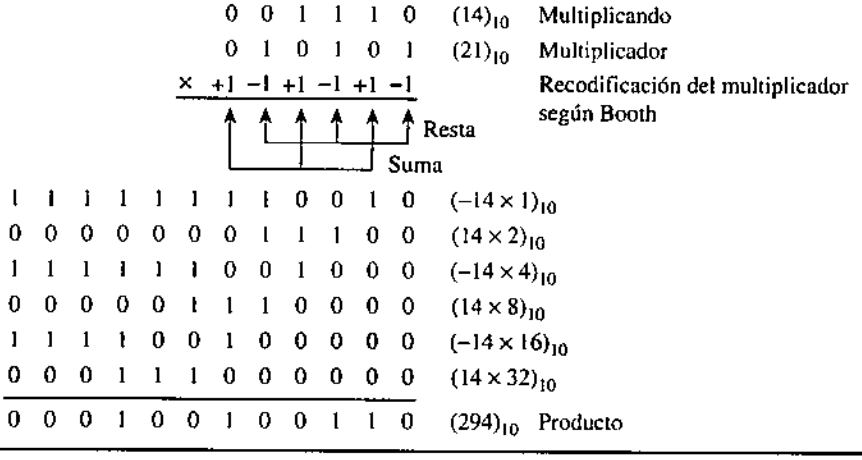


Figura 3.19 • Un ejemplo de ineficiencia en la aplicación del algoritmo de Booth.

El algoritmo de Booth modificado

Una solución a este problema consiste en agrupar los bits del multiplicador codificado en pares de bits, lo que lleva a la **codificación por pares de bits**, dando lugar a lo que se conoce como el **algoritmo de Booth modificado**. El agrupamiento de pares de bits de derecha a izquierda produce tres pares "+1, -1", tal como se observa en la figura 3.20. Debido a que el término +1 está a la izquierda del término -1, tiene un peso que es el doble del peso de la posición correspondiente al -1. Por consiguiente, puede pensarse en el par como si tuviese un valor conjunto de +2 – 1 = +1.

De forma similar, el par -1, +1 es equivalente a -2 + 1 = -1. Los pares +1, +1 y -1, -1 no pueden producirse. Se pueden generar un total de siete pares de bits diferentes, los que se muestran en la figura 3.21. En cada caso, se realiza el producto del multiplicando por el valor del par de bits codificado y el resultado obtenido se suma al producto parcial. En una implementación del sistema de codificación por pares de bits, se resuelven los dos

procedimientos de codificación en un solo paso, mediante la observación simultánea de tres bits del multiplicador, según se muestra en la tabla de bits correspondiente.

					0	0	1	1	1	0	$(21)_{10}$	Multiplicando
					0	1	0	1	0	1	$(14)_{10}$	Multiplicador
				×	+1	−1	+1	−1	+1	−1		Recodificación del multiplicador según Booth
					+1		+1		+1			Multiplicador codificado por pares de bits
0	0	0	0	0	0	0	0	1	1	1	0	$(14 \times 1)_{10}$	
0	0	0	0	0	0	1	1	1	0	0	0	$(14 \times 4)_{10}$	
0	0	0	0	1	1	1	0	0	0	0	0	$(14 \times 16)_{10}$	
0	0	0	1	0	0	1	0	0	1	1	0	$(294)_{10}$	Producto

Figura 3.20 • Producto que utiliza la codificación por pares de bits en el multiplicador.

Par de bits $(i+1, i)$		Par de bits recodificado (i)		Multiplicadores correspondientes $(i+1, i, i-1)$
0	0	=	0	000 o 111
0	+1	=	+1	001
0	−1	=	−1	110
+1	0	=	+2	011
+1	+1	=	—	
+1	−1	=	+1	010
−1	0	=	−2	100
−1	+1	=	−1	101
−1	−1	=	—	

Figura 3.21 • Codificación de pares de bits.

El proceso de codificar los bits del multiplicador de a pares garantiza que en el peor caso se realicen solo $w/2$ sumas (o restas) si el multiplicador está formado por w bits.

Multiplicadores matriciales

El método serie que fuera utilizado para multiplicar dos enteros no signados en la sección 3.2.1 requiere muy poca cantidad de circuitos, pero el tiempo necesario para realizar el producto de dos números de w bits crece proporcionalmente con w^2. Se puede acelerar el proceso de multiplicación de modo que se complete en solo $2w$ pasos por medio de la implementación del proceso manual de la figura 3.10, en paralelo. La idea general es la de formar productos de un bit entre cada bit del multiplicando y cada bit del multiplicador, y sumar luego cada fila de elementos del producto parcial, desde arriba hacia abajo en forma **sistólica** (fila por fila).

La estructura de un **multiplicador matricial** sistólico se ilustra en la figura 3.22. En la parte inferior de la figura se muestra un elemento para el cálculo de un producto par-

cial (PP). A través de la compuerta Y se realiza el producto de un bit del multiplicando (m_i) por un bit del multiplicador (q_j), lo que genera un producto parcial en la posición (i,j) de la matriz. Este producto parcial se suma con el producto parcial obtenido desde la etapa previa (b_j) y con cualquier arrastre que se hubiese generado en dicha etapa (a_j). El resultado tiene $2w$ bits y se lo obtiene en la parte inferior de la matriz (los w bits más significativos) y a la derecha de la misma (los w bits menos significativos).

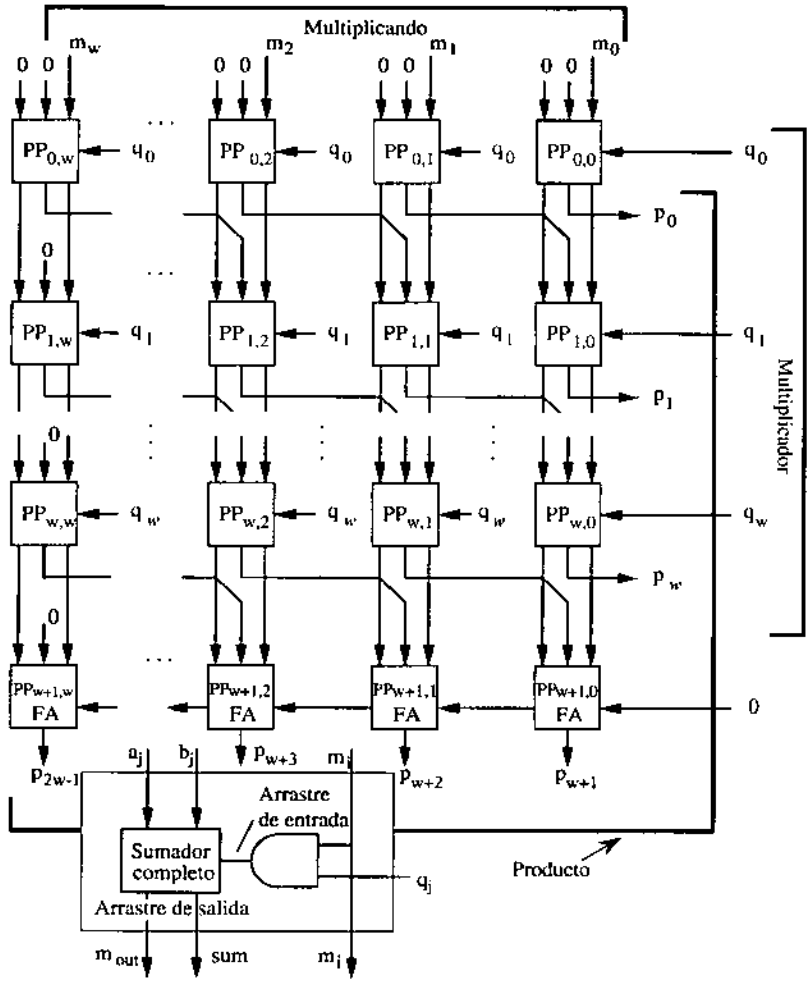


Figura 3.22 • Multiplicador paralelo de matriz segmentada.

3.5.3 División de alto rendimiento

La metodología de división de números enteros sin signo desarrollada en la sección 3.3.2 puede extenderse para permitir la generación de un resultado fraccionario cuando se calcula a/b. La idea general es la de escalar a y b para que parezcan enteros, realizar el pro-

ceso de división y, luego, volver a escalar el cociente para que se corresponda con el resultado correcto de la división de *a* por *b*.

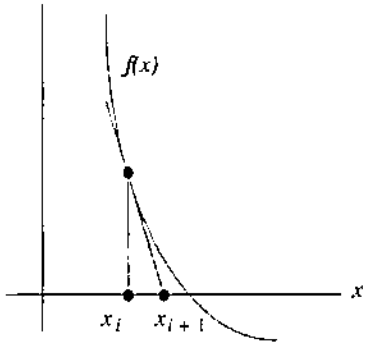


Figura 3.23 • Iteración de Newton para encontrar el cero (adaptado de D. Goldberg).

Otro método para realizar la división utiliza una tabla de búsquedas y un mecanismo de iteración. Uno de los métodos iterativos usados para encontrar las raíces de un polinomio se conoce como método de **iteración de Newton**, y se ilustra en la figura 3.23. El objetivo del método es encontrar el punto en que la función $f(x)$ cruza el eje x a través de una primera aproximación x_i, y luego refinar dicha aproximación utilizando el error entre $f(x_i)$ y cero.

La tangente en $f(x_i)$ puede representarse por la ecuación

$$y - f(x_i) = f'(x_i)(x - x_i)$$

La recta tangente cruza el eje x en

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

El proceso se repite mientras $f(x_i)$ se acerca a cero.

La cantidad de bits de precisión se duplica en cada iteración (véase D. Goldberg); por ende, si se requiere una precisión de 32 bits a partir de un único bit de precisión inicial, deben realizarse cinco iteraciones para llegar al resultado deseado. El problema ahora es cómo convertir la operación de división en una operación que permita encontrar un cero para $f(x_i)$.

Considérese la función $1/x - b$, que tiene un cero en $1/b$. Si se inicia con b, se puede calcular $1/b$ aplicando repetidamente el método de Newton. Dado que $f'(x) = -1/x^2$, se obtiene

$$x_{i+1} = x_i - \frac{1/x_i - b}{-1/x_i^2} = x_i + x_i - x_i^2 b = x_i\,(2 - x_i b)$$

Por consiguiente, solo se requiere la realización de productos y restas para poder resolver la división. Más aún, si la primera aproximación para determinar x_0 es lo suficientemente buena, puede ocurrir que solo hagan falta unas pocas iteraciones.

B = primeros 3 bits de b	Valor decimal real de 1/B	Entrada correspondiente en la tabla de búsquedas
0,100	2	10
0,101	1 3/5	01
0,110	1 1/3	01
0,111	1 1/7	01

Figura 3.24 • Tabla de búsqueda para el cálculo de x_0.

Antes de utilizar este método en un ejemplo, se hace necesario considerar cuál será la forma de intentar la aproximación inicial. Si se trabaja con fracciones normalizadas, es relativamente sencillo utilizar una tabla de búsqueda para los primeros dígitos. Considérese el caso de requerir el cálculo de 1/0,101101 utilizando una mantisa normalizada representada en 16 bits, sin la característica del primer uno implícito. Los primeros tres bits de cualquier mantisa binaria pueden adoptar cualquiera de los siguientes valores: 0,100, 0,101, 0,110 o 0,111. Estas fracciones corresponden, respectivamente, a los valores decimales 1/2, 5/8, 3/4 y 7/8. Las inversas de estos números son, respectivamente, 2, 8/5, 4/3 y 8/7. Los correspondientes valores binarios pueden almacenarse en una tabla de búsqueda y luego puede recuperarse x_0 sobre la base de los tres primeros dígitos de b.

El primer 1 de la mantisa no contribuye a la precisión, por lo que los tres primeros bits de la misma solo aportan 2 bits de precisión. Por lo tanto, la tabla de búsqueda solo requiere dos bits para cada entrada, tal como se indica en la figura 3.24.

Considérese ahora la división de 1/0,1011011 mediante esta misma representación de punto flotante. Se comienza encontrando x_0, para lo cual se utiliza la tabla de la figura 3.24. Los primeros tres bits de la fracción b son 101, lo que corresponde a $x_0 = 01$. Se puede calcular $x_1 = x_0 (2 - x_0 b)$ de donde se obtiene, en una operación sin signo binaria, 01(10 – (01)(0,1011011)) = 1,0100101. Los dos bits de precisión se han convertido en cuatro. Para este ejemplo, se busca retener la mayor precisión intermedia que permita el cálculo. En general, si se requiere un resultado de p bits, hace falta retener a lo sumo $2p$ bits de precisión intermedia. Al iterar nuevamente, se obtendrán ocho bits de precisión:

$$x_2 = x_1 (2 - x_1 b) = 1{,}0100\ 101\ (10 - (1{,}0100\ 101)(0{,}1011\ 011))$$

$$= 1{,}0110\ 0101\ 1001\ 0010\ 1110\ 1$$

Iterando nuevamente, se obtienen los 16 bits de precisión pretendidos:

$$x_3 = x_2(2 - x_2 b) =$$

$$= (1{,}0110\ 0101\ 1001\ 0010\ 1110)(1 - (1{,}0110\ 0101\ 1001\ 0010\ 1110)(1{,}1011\ 011))$$

$$= 1{,}0110\ 1000\ 0001\ 001 = (1{,}40652466)_{10}.$$

El valor exacto es $(1{,}40659341)_{10}$, pero la precisión obtenida es la máxima que puede lograrse con una representación de 16 bits.

3.5.4 Cálculo residual

Las operaciones de suma, resta y producto pueden realizarse en un único paso, sin necesidad de arrastres, utilizando el llamado **cálculo de residuos**. El sistema de numeración residual se basa en un conjunto de números enteros relativamente primos entre sí, llamados **módulos**. El residuo de un entero con respecto de un módulo particular es el menor resto entero positivo resultante de la división del entero dado por el módulo. Los números 5, 7, 9 y 4 forman un conjunto de módulos posibles. Con estos módulos pueden representarse en forma única la cantidad de 5 x 7 x 9 x 4 = 1260 enteros. La figura 3.25 muestra una tabla en la que se tiene la representación de los primeros veinte enteros decimales utilizando los módulos 5, 7, 9 y 4.

La suma y el producto desarrollados en el sistema numérico de residuos da por resultado números de residuos válidos, suponiendo que el tamaño del espacio numérico elegido sea lo suficientemente grande como para contener el resultado. La resta requiere que cada uno de los dígitos residuales del sustraendo sea complementado con respecto a su módulo antes de realizar la suma. La figura 3.26 ilustra ejemplos de suma y producto que utilizan a los números 5, 7, 9 y 4 como módulos. La suma se realiza en paralelo en cada columna, sin propagación de arrastre. El producto también se realiza en paralelo columna por columna, en forma independiente de las demás columnas.

Decimal	Residuo 5794	Decimal	Residuo 5794
0	0000	10	0312
1	1111	11	1423
2	2222	12	2530
3	3333	13	3641
4	4440	14	4052
5	0551	15	0163
6	1662	16	1270
7	2073	17	2381
8	3180	18	3402
9	4201	19	4513

Figura 3.25 • Representación de los veinte primeros enteros decimales en el sistema de residuos para los módulos propuestos.

29 + 27 = 56	
Decimal	Residuo 5794
29	4121
27	2603
56	1020

10 × 17 = 170	
Decimal	Residuo 5794
10	0312
17	2381
170	0282

Figura 3.26 • Ejemplos de suma y producto usando el sistema de residuos.

Si bien la aritmética de residuos puede ser muy rápida, trae aparejada una serie de desventajas. La división y la detección de signo suelen ser complejas, así como también la representación de fracciones. Las conversiones entre el sistema numérico de residuos y los sistemas numéricos posicionales conllevan cierta dificultad, y a menudo requieren la utilización de otros métodos como el **Teorema chino de los restos.** El problema de conversión es importante debido a que el sistema numérico de residuos no es demasiado útil si no se lo puede convertir en un formato posicional para realizar comparaciones de magnitudes. No obstante, para aquellas aplicaciones enteras en las que los tiempos requeridos por las operaciones de suma, resta y producto superan por mucho al tiempo utilizado para realizar divisiones, conversiones, etc., el sistema de residuos puede ser de aplicación práctica. Ejemplo de este caso es el de la multiplicación matriz-vector, ampliamente usada en el procesamiento de señales.

Ejemplo. Sumador de alto rendimiento para palabras largas

La utilización de sumadores con arrastre anticipado suele ser práctica para palabras de cuatro bits, no siendo tan útil su aplicación en palabras de 16 bits debido a las limitaciones eléctricas de entrada y salida de las compuertas lógicas que los conforman. La suma de dos números de 16 bits puede subdividirse en cuatro grupos de cuatro bits cada uno, los que se suman, en cada grupo, con sumadores con arrastre anticipado. La suma entre los distintos grupos de cuatro bits también se realiza con sumadores con arrastre anticipado. Esta forma de organizar la operación suele denominarse **sumador con arrastre anticipado entre grupos** (GCLA, *group carry lookahead adder*). En este ejemplo se compararán, en términos de retardos y cargas eléctricas, un sumador de 16 bits con arrastre anticipado (CLA) y otro que utiliza la técnica de arrastre anticipado entre grupos (GCLA).

La figura 3.27 muestra un sumador de 16 bits con arrastre anticipado entre grupos, compuesto por cuatro sumadores de cuatro bits, y alguna lógica adicional que pueda generar los arrastres entre los distintos sumadores de cuatro bits.

Cada grupo se comporta como un sumador convencional, excepto por el hecho de que el arrastre que ingresa al bit menos significativo de cada grupo debe ser tratado como una variable en lugar de ser cero, originándose señales de **generación grupal** (*GG*) y **propagación grupal** (*GP*). Se genera una señal *GG* cuando se produce un arrastre en alguna parte dentro de un grupo y si todas las señales de propagación más significativas son ciertas.

Esto significa que la señal de arrastre ingresada a un grupo se propagará a todo lo largo del mismo. Las ecuaciones correspondientes a las señales *GG* y *GP* menos significativas de la figura 3.27 son las que siguen:

$$GG_0 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G$$

$$GP_0 = P_3P_2P_1G_0$$

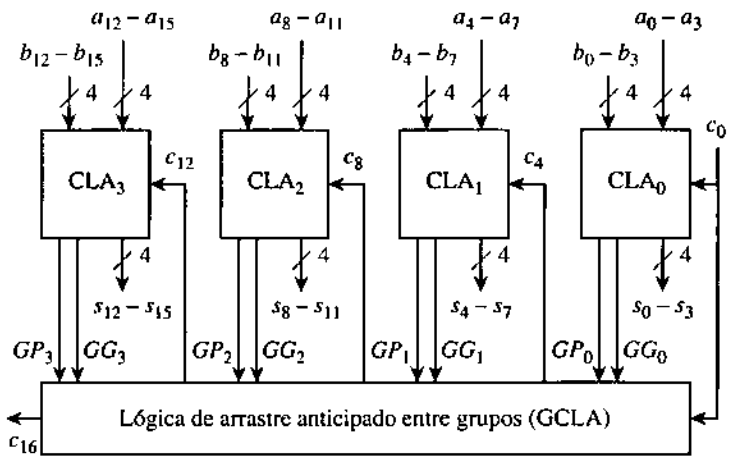


Figura 3.27 • Sumador de 16 bits con arrastre anticipado entre grupos.

La determinación de las señales *GG* y *GP* restantes se realiza en forma similar.

El arrastre que ingresa en cada grupo, excepto para el caso del arrastre inicial, se determina desde las señales *GG* y *GP*. Por ejemplo, c_4 es cierta cuando GG_0 es cierta o cuando GP_0 y c_0 son ambas ciertas. La ecuación correspondiente es:

$$c_4 = GG_0 + GP_0\,c_0$$

Los arrastres de orden superior que salen desde cada grupo se calculan en forma similar:

$$c_8 = GG_1 + GP_1c_4 = GG_1 + GP_1GG_0 + GP_0c_0$$
$$c_{12} = GG_2 + GP_2c_8 = GG_2 + GP_2GG_1 + GP_2GP_1GG_0 + GP_2GP_1GP_0c_0$$
$$c_{16} = GG_3 + GP_3c_{12} = GG_3 + GP_3GG_2 + GP_3GP_2GG_1 +$$
$$GP_3GP_2GP_1GG_0 + GP_3GP_2GP_1GP_0c_0$$

En términos de retardos de compuertas, un sumador con arrastre anticipado de 16 bits tiene como camino más largo para sus señales lógicas el equivalente a cinco niveles de compuertas, requerido para generar el bit más significativo de la suma, tal como se planteara

en la sección 3.5.1. Cada uno de los sumadores individuales en el sumador grupal tiene también, al menos, cinco niveles de compuertas en su trayecto más largo. Las señales *GG* y *GP* se generan con tres niveles de compuertas, y las señales de arrastre que salen de cada uno de los grupos también se generan en dos niveles de compuertas, con lo que se obtiene un total de cinco niveles de compuertas como retardo para la generación del arrastre que sale de cada uno de los grupos. En la posición más significativa (s_{15}), se requieren cinco niveles para generar c_{12} y otras cinco compuertas para generar s_{15}, dando como resultado que la peor solución presenta diez niveles de retardo a lo largo del sumador grupal.

Con respecto a las entradas y salidas, la cantidad máxima de entradas en cualquier compuerta de un sumador de cuatro bits con arrastre anticipado es cuatro, según puede verse en la figura 3.17. En general, la máxima cantidad de entradas en una compuerta de un sumador con arrastre anticipado de n bits es n. Así, la máxima cantidad de entradas de cualquier compuerta en un sumador de 16 bits con arrastre anticipado es de 16. Como comparación, la máxima cantidad de entradas en un sumador grupal de 16 bits es solo de 5, en la generación de c_{16}. Las cantidades de salidas son las mismas que las de entradas.

En resumen, un sumador de arrastre anticipado de 16 bits tiene solo la mitad de la cantidad de niveles lógicos que el sumador grupal de 16 bits (5 niveles de compuertas contra 10 niveles en el segundo caso). La mayor cantidad de entradas de un sumador con arrastre anticipado de 16 bits es de 16, lo que es más de tres veces mayor que la característica similar en un sumador grupal (16 contra 5). Las características de salida son las mismas que las de entrada en cada caso. •

3.6 Estudio de un caso: calculadora que utiliza el sistema decimal codificado en binario

El cálculo aritmético en máquinas calculadoras se resuelve tradicionalmente en el sistema decimal, en lugar de hacerlo en el sistema binario. Las calculadoras deben ser de tamaño reducido y económicas, y por esa razón los números decimales se representan en BCD (decimal codificado en binario, véase el capítulo 2), utilizando cuatro bits por dígito, en lugar de usar el sistema binario de numeración, que demanda un proceso de conversión entre sistemas con altas necesidades de recursos. Para realizar las operaciones podrá pensarse en una unidad aritmético-lógica pequeña, de 4 bits, que realice los cálculos en serie, operando sobre los dígitos codificados BCD, uno tras otro.

3.6.1 La calculadora HP9100A

La calculadora HP9100A, que apareció y se hizo popular sobre el final de la década de 1960, realizaba las funciones aritméticas básicas: suma, resta, producto y cociente, así como radicación, e^x, $\ln x$, $\log x$, funciones trigonométricas y otras, utilizando en todos los casos aritmética decimal. La calculadora HP9100A es, en realidad, una calculadora de escritorio (véase

la figura 3.28), considerada pequeña para lo que hacía con la tecnología de aquellos días. En el visor de la HP9100 se muestran diez dígitos significativos, pero todos los cálculos se realizaban con 12 dígitos significativos, donde los dos últimos dígitos (**dígitos de guarda**) eran utilizados para administrar los errores de redondeo y truncado. Si bien la HP9100A puede parecer hoy una reliquia, los métodos de operación aritmética siguen siendo relevantes.

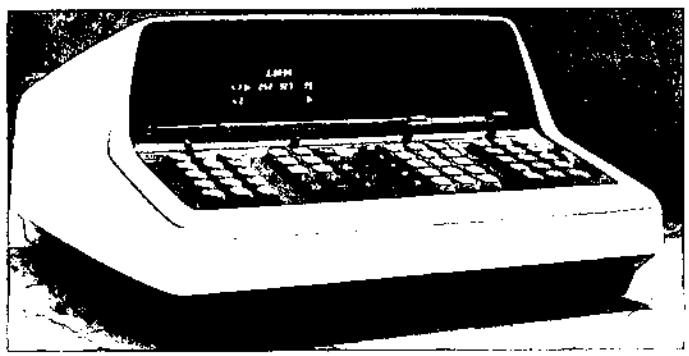


Figura 3.28 • Calculadora de escritorio de la serie HP9100A. Fuente: Museo de calculadoras HP, http://www.hpmuseum.org.

Las dos secciones siguientes describen técnicas genéricas para la realización de operaciones de suma y resta entre números codificados en BCD, tanto en punto fijo como en punto flotante. Las demás operaciones descriptas en las secciones restantes se realizan en forma similar, haciendo uso de las operaciones de suma y resta.

3.6.2 Suma y resta de números codificados en BCD

Considérese la suma de $(+255)_{10}$ y $(+63)_{10}$, codificados en BCD, de acuerdo con lo ilustrado en la figura 3.29. Cada dígito decimal ocupa cuatro bits, y la suma se realiza dígito por dígito (no bit por bit), de derecha a izquierda, tal como se procedería cuando se realizan operaciones decimales a mano. El resultado $(+318)_{10}$, se obtiene en formato BCD, según se observa en la figura.

	0	1	0	0	← Arrastres
	0000	0010	0101	0101	$(+255)_{10}$
	$(0)_{10}$	$(2)_{10}$	$(5)_{10}$	$(5)_{10}$	
+	0000	0000	0110	0011	$(+63)_{10}$
	$(0)_{10}$	$(0)_{10}$	$(6)_{10}$	$(3)_{10}$	
	0000	0011	0001	1000	$(+318)_{10}$
	$(0)_{10}$	$(3)_{10}$	$(1)_{10}$	$(8)_{10}$	

Figura 3.29 • Ejemplo de suma que utiliza el sistema decimal codificado en binario (BCD).

La resta BCD se resuelve en forma similar a la forma en que se la ejecuta en la representación binaria de complemento a dos (utilizando el complemento del sustraendo), con la única diferencia de que se debe usar el complemento a diez en lugar del complemento a dos. Considérese la resta entre $(255)_{10}$ y $(63)_{10}$, = $(192)_{10}$. Esta operación se puede convertir en la suma equivalente $(255 + (-63) = 192)_{10}$. Se comienza calculando el complemento a 9 de 63:

$$9999 - 0063 = 9936$$

al cual se le suma 1 para obtener el complemento a 10 del mismo valor:

$$9936 + 1 = 9937$$

A continuación se realiza la suma, tal como se ilustra en la figura 3.30. Nótese que al realizar la suma, del mismo modo que en el caso del complemento a 2, se descarta el bit más significativo.

	1	1	0	1	0	← Arrastres
		0000	0010	0101	0101	$(+255)_{10}$
+		1001	1001	0011	0111	$(-63)_{10}$
	1	0000	0001	1001	0010	$(+192)_{10}$
	↑					
	Descartar arrastre					

Figura 3.30 • Suma BCD en notación de complemento a dos.

A diferencia de la representación en complemento a dos, no se puede determinar el signo del resultado simplemente mirando el bit más significativo. En el complemento a diez, el número es positivo si el dígito más significativo está en el rango entre 0 y 4, incluidos los extremos, y es negativo en caso contrario. (Las representaciones BCD del 4 y el 5 son, respectivamente, 0100 y 0101; ambas tienen el bit más significativo en cero y, sin embargo, una representa un número positivo y la otra representa un número negativo.) Si se utiliza una representación de exceso 3 para cada dígito, entonces sí se podrá considerar al bit más significativo como bit de signo. La figura 3.31 representa ambas codificaciones. Debe notarse que seis de las diez combinaciones de bits no pueden producirse, por lo que se las define como combinaciones **redundantes*** (*don't cares*), indicadas con "d" en dicha figura.

* *N. de T.*: La terminología inglesa, que se refiere a "estados que no interesan", suele traducirse de varias formas diferentes, tales como "combinaciones indeterminadas", "redundantes", "irrelevantes", etcétera.

Combinación BCD	Valor BCD 8421	Representación exceso 3	
0 0 0 0	0	d	
0 0 0 1	1	d	
0 0 1 0	2	d	
0 0 1 1	3	0	
0 1 0 0	4	1	Números positivos
0 1 0 1	5	2	
0 1 1 0	6	3	
0 1 1 1	7	4	
1 0 0 0	8	5	
1 0 0 1	9	6	Números negativos
1 0 1 0	d	7	
1 0 1 1	d	8	
1 1 0 0	d	9	
1 1 0 1	d	d	
1 1 1 0	d	d	
1 1 1 1	d	d	

Figura 3.31 • Codificación de dígitos BCD en código exceso 3.

Considérese ahora el diseño de un sumador completo BCD. Este sumador deberá sumar dos dígitos BCD y un arrastre de entrada, debiendo producir como resultado un dígito BCD y un arrastre de salida, todo codificado en notación exceso 3. La figura 3.32 muestra un diseño que utiliza sumadores completos binarios de complemento a dos. Los dos dígitos codificados en exceso 3 se suman en los cuatro sumadores de un bit del bloque superior de la figura. Dado que cada dígito está expresado en exceso 3, el resultado termina expresado en exceso 6. Para volver a llevar el resultado a la representación en exceso 3, se requiere restar 3 al resultado. Como alternativa, se puede sumar 13 al resultado, dado que 16 – 3 = 16 + 13 si se trabaja en cuatro bits, y se debe descartar el bit de arrastre que se genera luego del bit más significativo. El sumador de la figura 3.32 utiliza esta última técnica, por lo que adiciona $13_{10} = 1101_2$ al resultado de la primera suma. Nótese que esto funciona solo si no hay arrastre en la primera operación. Si se produce un arrastre, se requiere restarle 10 al resultado (o, lo que es lo mismo, sumarle 6), además de la resta de 3 (o suma de 13) necesaria para recuperar la representación en exceso 3 y generar el arrastre final. El criterio utilizado para este caso es el de sumar $3_{10} = 0011_2$, lo que produce el mismo efecto que sumar (6 + 13) % 16 = 3, de acuerdo con lo que se puede ver en la figura 3.32.[1]

Si se requiere realizar una resta en BCD, se puede crear un circuito restador completo en complemento a diez usando sumadores completos decimales, tal como fue plantea-

1. El operador %, utilizado en el lenguaje de programación C, devuelve el resto de la división entre dos operandos.

do en el caso del restador en complemento a dos descripto en la sección 3.2.2. Como alternativa, se puede generar el complemento a diez del sustraendo y aplicar luego los métodos comunes de la suma BCD. La figura 3.33 ilustra la operación $(21 - 34 = -13)_{10}$, usando el segundo método mencionado para la resta de números de cuatro dígitos. El complemento a diez de 34 se suma con el minuendo 21, dando por resultado el valor 9987 en complemento a 10, lo que equivale a (-13) en magnitud y signo.

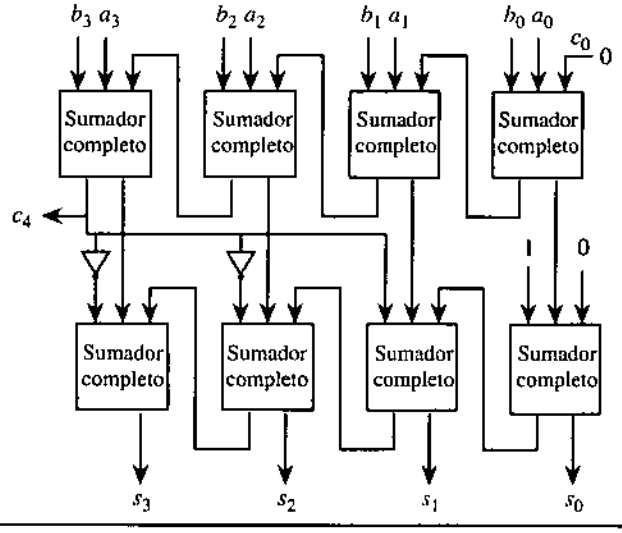


Figura 3.32 • Un sumador completo BCD.

0021	0021
+ 9966	- 0034
9987	- 0013
Complemento a diez	Magnitud y signo

Figura 3.33 • El cálculo de (21 – 34 = -13) en complemento a dos y en magnitud y signo.

3.6.3 Suma y resta en BCD punto flotante

Considérese una representación decimal de punto flotante con un exponente de dos dígitos representado en magnitud y signo, y una mantisa signada de ocho dígitos. En una calculadora, la correspondiente entrada de datos podría verse como

$$-0{,}37100000 \times 10^{-12}$$

representada en su forma normalizada.

¿Cómo se almacena el número en realidad? Un usuario de calculadora ve magnitud y signo tanto para el exponente como para la mantisa, aunque internamente pueda usar-

se una representación de complemento a diez tanto para el exponente como para la mantisa. Para el caso anterior, la representación del exponente en notación de complemento a diez sería 88, en tanto que la mantisa se representaría como 62900000. Si se utiliza una representación binaria en exceso 3 se obtendría 1011 1011 como valor para el exponente, y una mantisa representada como 1001 0101 1100 0011 0011 0011 0011 0011. Nótese que, dado que el bit más significativo debe considerarse como signo, el rango del exponente es [+50 .. -49], en tanto que el de la mantisa es [-0,50000000 .. + 0,49999999].

Si se trata de representar ahora el número + 0,9 en base 10, vuelve a aparecer un inconveniente debido a que el bit más significativo de la mantisa se usa como bit de signo. Esto significa que no puede utilizarse 1100 como dígito más significativo de la mantisa, dado que, si bien corresponde a la representación exceso 3 del número 9, hace que la mantisa aparezca como si fuese negativa. Existe una solución mejor. Simplemente, es cuestión de usar la notación de complemento a 10 para la aritmética entera, tal como en el caso de los exponentes, y usar magnitud y signo para las mantisas.

El resumen hasta el momento es el siguiente: se utiliza una representación de complemento a diez para el exponente debido a que se trata de un entero, y se utiliza una representación decimal de magnitud y signo para la mantisa. Se mantiene un bit de signo separado para la mantisa, de modo que cada dígito pueda adoptar cualquiera de los diez valores que van de 0 a 9 (excepto para el primer dígito, que no puede ser cero) y así podrá representarse el +0,9. También deberían representarse los exponentes en exceso 50 para que las comparaciones se resuelvan más fácilmente. El ejemplo anterior, si se lo mira internamente, adopta la siguiente forma, en representación binaria de exceso 3 con un exponente de dos dígitos representado en exceso 50:

Bit de signo:	1
Exponente:	0110 1011
Mantisa:	0110 1010 0100 0011 0011 0011 0011 0011 0011

Para sumar dos números representados en este formato, se debe proceder a recorrer los mismos pasos que se llevaran a cabo en las representaciones binarias de punto flotante planteadas anteriormente. Se comienza la operación ajustando el exponente y la mantisa del menor de los operandos hasta que ambos exponentes coincidan. Si la diferencia entre los exponentes es tan grande como para que la mantisa del menor de los operandos se haya desplazado totalmente hacia la derecha, dicho operando debe tratarse como si valiese 0. Luego de ajustar la mantisa del operando menor, se convierten uno o ambos de su formato de magnitud y signo al formato de complemento a diez, se trate de una suma o una resta y de acuerdo con el signo de los respectivos operandos. Nótese que no habrá inconvenientes de funcionamiento dado que las mantisas están siendo tratadas como enteros.

Resumen

Los cálculos aritméticos dentro de una computadora pueden efectuarse tal como se realizan en forma manual las operaciones decimales, siempre y cuando se tome en cuenta el sistema de numeración en que se está operando. Para la representación de números enteros se usan, habitualmente, las de complemento a dos o a diez, en tanto que los valores fraccionarios se representan en magnitud y signo debido a la dificultad de manejar fracciones positivas y negativas de modo uniforme.

La eficiencia del cálculo puede mejorar si se utilizan las técnicas de codificación de Booth y del apareamiento de bit para saltear cadenas de unos. Una alternativa para mejorar el rendimiento se basa en el uso de los métodos que permiten sumar sin arrastres, como es el caso de la aritmética de residuos. Si bien la suma sin restos puede ser el enfoque más rápido en lo que hace a las complejidades relacionadas con los tiempos y los circuitos, en la práctica se utilizan los códigos pesados más comunes con el objeto de simplificar las comparaciones y la representación de las mantisas.

Para lectura posterior

D. Goldberg es una fuente concisa pero profunda para diversos aspectos de la aritmética de computadoras. V. C. Hamacher y otros provee el tratamiento clásico de la aritmética de enteros. M. J. Flynn ofrece un tratamiento acerca de la detección de la división por cero. H. L. Garner realiza una descripción completa del sistema de numeración residual en tanto que I. Koren brinda un tratamiento tutorial del mismo tema. A. Huang y J. W. Goodman describen el proceso de construcción de un procesador residual basado en memoria. Koren también ofrece detalles adicionales acerca de la utilización de unidades con arrastre anticipado en forma encadenada. D. S. Cochran es una buena fuente para la programación de la calculadora HP9100A.

Cochran, D. S., "Internal Programming of the 9100A Calculator", en: *Hewlett Packard Journal*, septiembre de 1968; véase también http://www.hpmuseum.org/journals/9100:prg.htm.

Flynn, M. J., "On Division by Functional Iteration", en: *IEEE Trans. Comp.*, C.19, n° 8, agosto de 1970, p.p. 702-706.

Garner, H. L., "The Residue Number System", en: *IRE Transactions on Electronic Computers*, vol. 8, junio de 1959, p.p. 140-147.

Goldberg, D., "Computer Arithmetic", en: D. A. Patterson y J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2ª ed., Morgan Kaufmann, 1995.

Hamacher, V. C., Z. G. Vranesic y S. G. Zaky, *Computer Organization*, 3ª ed., McGraw-Hill, 1990.

Huang A. y J. W. Goodman, "Number Theoretic Processors, Optical and Electronic", en: *SPIE Optical Processing Systems*, vol. 185, 1979, p.p. 28-35.

Koren I., *Computer Arithmetic Algorithms*, Prentice Hall, 1993.

Problemas

3.1 Indicar el resultado que se obtiene al sumar los siguientes pares de números de cinco bits, representados en complemento a dos con un bit de signo y cuatro de valor numérico, indicando en cada caso la ocurrencia o no de desborde.
10110 + 10111 =
11110 + 11101 =
11111 + 01111 =

3.2 Una forma de determinar la ocurrencia de un desborde en la suma de dos números consiste en detectar que el resultado de la suma de dos números positivos es negativo o que el resultado de sumar dos números negativos es positivo. Las reglas para la resta son diferentes: existe desborde cuando al restar un número negativo de uno positivo se obtiene un resultado negativo, o si el resultado de restar un número positivo de otro negativo da un resultado positivo. Calcular la resta de los números que se indican y determinar si se ha producido desborde o no. La resta debe calcularse sin considerar el complemento a dos del sustraendo, sino restando bit por bit y analizando los arrastres producidos en cada posición.
0101 – 0110 =

3.3 Sumar los dos números binarios 1011,101 + 0111,011 considerando que los mismos están representados
a. en complemento a dos.
b. en complemento a uno.
Para cada caso, indicar si hay desborde.

3.4 Mostrar el proceso a realizar para obtener el producto sin signo de 1010 x 0101, utilizando el esquema de la figura 3.12.

3.5 Mostrar el proceso a realizar para obtener el producto no signado de 11,1 (multiplicando) por 01,1 (multiplicador) mediante el tratamiento de los operandos como enteros. El resultado a obtener debería ser 101,01.

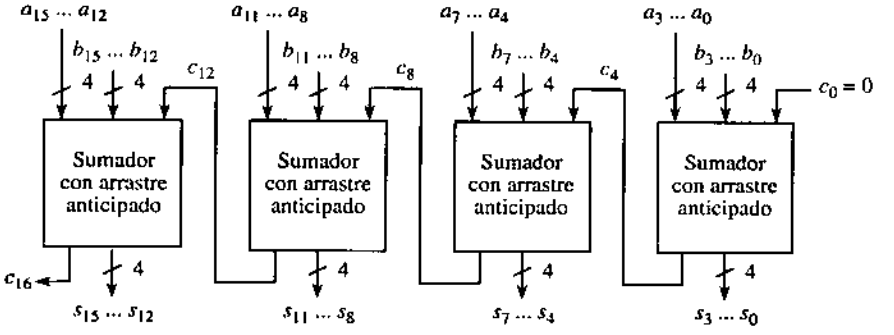
3.6 Mostrar el proceso a realizar para obtener el cociente sin signo de 1010 ÷ 0101, utilizando el esquema de la figura 3.15.

3.7 Mostrar el proceso a realizar para obtener el cociente sin signo de 1010 ÷ 0100, continuando la división para obtener el resultado con parte fraccionaria. El resultado a obtener debería ser $10{,}1_2$.

3.8 La ecuación utilizada en la sección 3.5.1 para obtener el valor de c_4 en un sumador con arrastre anticipado supone que en el proceso de suma c_0 vale 0. Si se realiza una resta utilizando el

circuito sumador-restador de la figura 3.6, c_0 vale 1. Rescribir la ecuación que define c_4 cuando $c_0 = 1$.

3.9 El sumador de 16 bit de la figura utiliza arrastre serie entre cuatro sumadores de cuatro bits con arrastre anticipado.



a. ¿Cuál es el máximo retardo, medido en cantidad de niveles de compuertas, a lo largo del sumador?

b. ¿Cuál es el mínimo retardo a lo largo del sumador, entre cualquier entrada y cualquier salida?

c. ¿Cuál es el retardo en la salida s_{12}?

3.10 Utilizar el algoritmo de Booth (sin codificación de pares) para multiplicar 010011 (multiplicando) por 011011 (multiplicador).

3.11 Repetir el producto anterior utilizando codificación de pares de bits.

3.12 Calcular el máximo retardo, en niveles de compuertas, que se produce a lo largo de un sumador de 32 bits con arrastre anticipado.

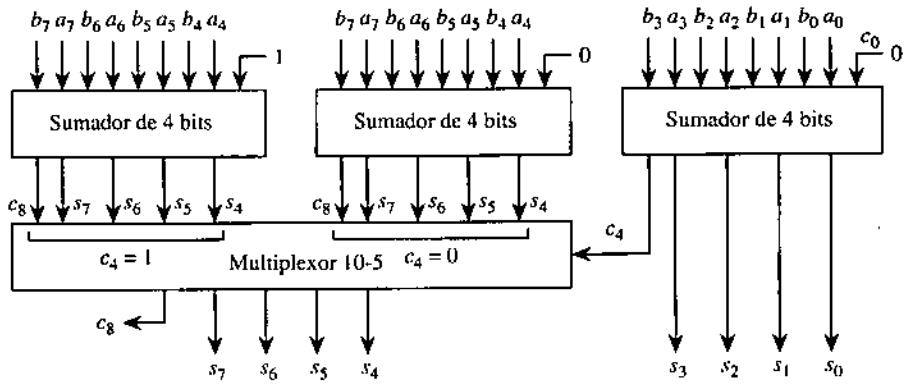
3.13 ¿Cuál es el máximo número de entradas para cualquier compuerta lógica en un sumador de 32 bits con arrastre anticipado, si se usa el esquema planteado en este capítulo?

3.14 En un **sumador con selección de arrastre** (*carry select adder*) el arrastre se propaga desde una etapa del sumador, en forma similar pero no exactamente igual a la que se utiliza en un sumador con arrastre anticipado. Como en muchos otros sumadores, la salida de arrastre de una etapa dada puede ser 0 o 1. En un sumador con selección de arrastre, para cada etapa del sumador se realizan dos sumas en paralelo. Una de ellas supone una entrada de arrastre de valor 0, y la otra supone un arrastre de entrada cuyo valor es 1. Se realiza luego (por ejemplo, con un multiplexor), la selección de cuál de las dos sumas debe utilizarse. El esquema básico se muestra en la figura siguiente para un sumador de ocho bits con selector de arrastre.
Suponiendo que cada uno de los sumadores de cuatro bits utiliza internamente la técnica de arrastre anticipado, comparar la cantidad de retardos o niveles de compuertas requeridas pa-

ra sumar dos números de ocho bits, si dichos sumadores de cuatro bits utilizan el esquema de selección de arrastre, contra el valor resultante si el arrastre se trasporta en serie desde un sumador de cuatro bits al otro.

a. Dibujar un diagrama de un circuito sumador de ocho bits que utilice la configuración de arrastre anticipado y que use como bloques los sumadores de cuatro bits de la figura.

b. Calcular la cantidad de niveles de compuertas requeridas para cada una de las configuraciones planteadas, tanto para el sumador con selección de arrastre como para el sumador diseñado en la pregunta a.



3.15 El camino con mayor número de niveles de compuertas a lo largo del multiplicador matricial de la figura 3.22 se inicia en el elemento PP ubicado arriba a la derecha, recorre la columna hasta la fila inferior y, finalmente, cruza hacia la izquierda. El máximo número de niveles de compuertas en un elemento PP es tres. ¿Cuántos niveles de compuertas hay en el camino de máximo retardo de un multiplicador matricial que produce un resultado de p bits?

3.16 Se tienen unidades multiplicadoras que producen productos binarios sin signo de 16 bits a partir de dos entradas sin signo de 8 bits. Se tienen, además, sumadores completos de 16 bits que producen una suma de 16 bits y una salida de arrastre a partir de dos entradas de 16 bits y un arrastre de entrada. Conectar los bloques mencionados de modo tal que el conjunto obtenido multiplique dos números sin signo de 16 bits y entregue un resultado de 32 bits.

3.17 En una división se desea obtener una precisión de 32 bits, usando el método iterativo de Newton. Si se utiliza una tabla de búsquedas que provee 8 bits de precisión para el intento inicial, ¿cuántas iteraciones posteriores se requieren?

3.18 Sumar los números $(641)_{10}$ y $(259)_{10}$ en BCD sin signo, utilizando la menor cantidad de dígitos posibles en el resultado.

3.19 Sumar $(123)_{10}$ y $(-178)_{10}$ en BCD con signo, usando palabras de cuatro dígitos.

Capítulo 4

La arquitectura de programación

A lo largo de este capítulo se encara un tema fundamental en la arquitectura de computadoras: el lenguaje que puede entender el hardware, conocido como **lenguaje de máquina**. El lenguaje de máquina se analiza habitualmente en función del llamado **lenguaje ensamblador** (*assembly language*) o lenguaje **simbólico**, funcionalmente equivalente al lenguaje de máquina correspondiente excepto por el hecho de que utiliza nombres algo más intuitivos como, *Move*, *Add* y *Jump**, en reemplazo de las palabras binarias que en realidad emplea el lenguaje de máquina. (Para los programadores es mucho más clara y menos factible de error una construcción tal como "Add r0, r1, r2" que 0110 1011 1010 1101.)

El desarrollo del capítulo comienza por la descripción de la visión de la computadora y de sus operaciones desde su **arquitectura de programación** (ISA, *Instruction Set Architecture*). El enfoque desde la arquitectura de programación de una máquina se corresponde con el nivel del lenguaje ensamblador y del código de máquina descripto en la figura 1.4. Se halla entre el enfoque desde el nivel del lenguaje de alto nivel, en el cual poco o nada se hace visible del esquema circuital de la computadora, y el nivel del control, en el cual las instrucciones de la máquina se interpretan como acciones de transferencias entre registros al nivel de las unidades funcionales.

Con el objeto de describir la naturaleza del lenguaje simbólico y la programación en dicho lenguaje, se utilizará como modelo de arquitectura la de la computadora ARC, simplificación de la arquitectura comercial SPARC, común en las computadoras Sun. (Sobre otros modelos de arquitectura adicionales, se puede consultar en *The Computer Architecture Companion*. Véase http://www.pearsonedlatino.com/murdocca.)

La utilidad de los distintos tipos de instrucciones se ilustra con ejemplos prácticos de programación en lenguaje ensamblador, completándose el tema con un caso de estudio referido a los códigos de Java como ejemplo de un lenguaje ensamblador común, portable, que puede implementarse utilizando los códigos naturales de otra máquina.

---

* *N. de T.*: Dado que la mayoría de los lenguajes simbólicos utilizan nombres asociados con el idioma inglés para representar las instrucciones, se ha optado por ejemplificar respetanto las siglas originales.

4.1. Componentes circuitales de la arquitectura de programación

La arquitectura de programación de una computadora coloca a quien programa en lenguaje ensamblador ante una visión de la máquina que incluye todos los elementos circuitales accesibles al programador y las instrucciones que manejan información dentro del hardware. Esta sección analiza los componentes de hardware tal como se los ve desde el enfoque del programador de lenguaje de máquina. Comienza con el análisis del sistema como un conjunto, con la unidad de proceso interactuando con la memoria principal y realizando operaciones de entrada-salida con el mundo real.

4.1.1 Una revisión del modelo de bus

La figura 4.1 intenta una revisión del modelo de computadoras basado en buses, que fuera planteado en el capítulo 1. El propósito del bus es el de reducir la cantidad de interconexiones entre la CPU y sus subsistemas. En lugar de tener caminos de comunicación diferentes con la memoria y con cada uno de los dispositivos de entrada-salida, la CPU se interconecta con la memoria y con los sistemas de entrada y salida a través del **bus del sistema**. En sistemas más complejos puede haber buses separados: por un lado, entre la CPU y la memoria, y por otro, entre la CPU y los dispositivos de entrada-salida.

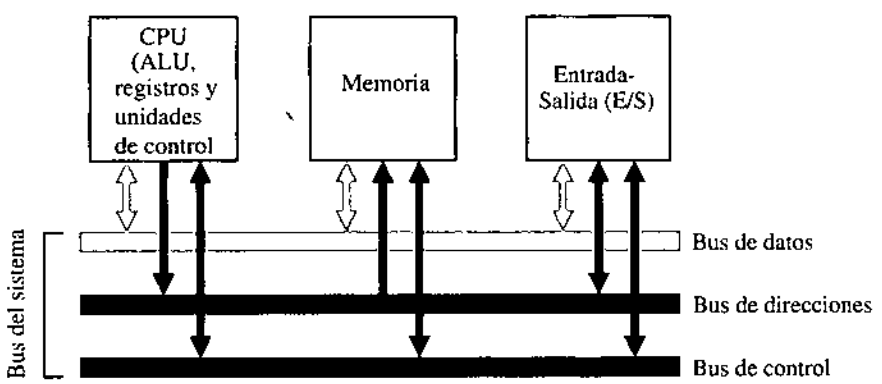


Figura 4.1 • El modelo de un sistema de computación con estructura de bus.

No todos los componentes del sistema se conectan al bus de la misma forma. La CPU genera las direcciones que se transfieren sobre el bus de direcciones, mientras que la memoria recibe esas direcciones a través del bus de direcciones. La memoria nunca genera direcciones, así como la CPU nunca recibe direcciones, por consiguiente, no hay interconexión en ese sentido.

En un escenario típico, el usuario escribe un programa en lenguaje de alto nivel, el que se traduce a lenguaje ensamblador por medio de un programa compilador. Luego, un programa ensamblador convierte el programa en lenguaje simbólico a lenguaje de má-

quina, para su almacenamiento en disco. Como paso previo a su ejecución, el sistema operativo de la computadora carga el programa en lenguaje de máquina desde el disco a la memoria principal.

Durante la ejecución del programa cada instrucción se carga en la CPU desde la memoria, a razón de una instrucción por vez, junto con cualquier dato que sea necesario para ejecutar la instrucción. La salida del programa se coloca en un dispositivo, tal como una unidad de disco o una pantalla de video. Todas estas operaciones están reguladas, por la unidad de control, la que será analizada en detalle en el capítulo 6. La comunicación entre los tres componentes (CPU, memoria y entrada-salida) se maneja por medio de buses.

Debe quedar claro que las instrucciones se ejecutan dentro de la CPU, a pesar de que las instrucciones y los datos se encuentran almacenados en memoria. Esto significa que las instrucciones y los datos deben cargarse desde la memoria hacia los registros de la CPU, en tanto que los resultados deben devolverse a la memoria para su almacenamiento desde los registros de la CPU.

4.1.2 Memoria

La memoria de una computadora consiste en un conjunto de registros numerados (direccionados) en forma consecutiva, cada uno de los cuales normalmente almacena un byte de información. Un **byte** es un conjunto (llamado habitualmente **octeto** por quienes trabajan en comunicaciones digitales) de ocho bits. Cada registro tiene una dirección, a la que se suele designar como **locación de memoria.** La denominación ***nibble*** (también se escribe como ***nybble***) se refiere a un conjunto de cuatro bits. Normalmente, hay acuerdo acerca de los significados de los términos "bit", "byte" y "nibble", pero no así sobre el concepto de **palabra**, el que depende de la arquitectura particular de cada procesador. Los tamaños de palabra típicos son de 16, 32, 64 y 128 bits, siendo el tamaño de palabra de 32 bits el más común para las computadoras usadas principalmente hoy en día, mientras se observa el crecimiento de la popularidad de las palabras de 64 bits. En este texto, se supone que las palabras tendrán 32 bits a menos que se especifique otra cosa. La figura 4.2 compara estos formatos de datos.

En una máquina con capacidad de direccionamiento al byte, el menor objeto al que puede hacerse referencia en memoria es el byte; no obstante, es habitual que tenga instrucciones que pueden leer y escribir palabras de más de un byte. Las palabras de más de un byte se almacenan como una secuencia de bytes, y se direccionan a partir del byte menos significativo de la palabra almacenada. La mayor parte de las computadoras actuales poseen instrucciones que pueden acceder a distintos tamaños de datos.

Cuando se utilizan palabras de más de un byte, hay dos alternativas en cuanto a la forma de almacenar sus bytes en memoria; en la primera de ellas, el byte más significativo se almacena en la dirección más baja de memoria (***big endian***) y en la otra, el byte menos significativo es el que se almacena en la dirección más baja de memoria (***little***

Bit	0
Nibble	0110
Byte	10110000
Palabra de 16 bits (media palabra)	11001001 01000110
Palabra de 32 bits	10110100 00110101 10011001 01011000
Palabra de 64 bits (doble)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101
Palabra de 128 bits (cuádruple)	01011000 01010101 10110000 11110011 11001110 11101110 01111000 00110101 00001011 10100110 11110010 11100110 10100100 01000100 10100101 01010001

Figura 4.2 • Tamaños comunes para los formatos de palabra de datos.

endian). El término "endian" proviene de discutir si los huevos deben romperse por su extremo más grande o por el más pequeño, lo que provocó una guerra entre políticos pendencieros en el afamado cuento de Jonathan Swift, *Los viajes de Gulliver*. La figura 4.3 ilustra los formatos mencionados para una palabra de 4 bytes, 32 bits.

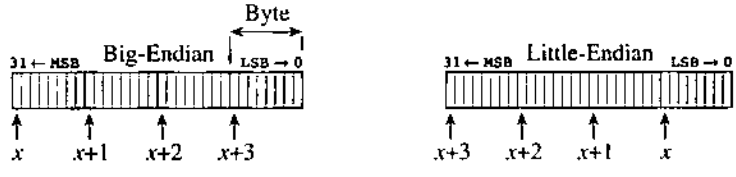


Figura 4.3 • Formatos de almacenamiento *big endian* y *little endian*.

La estructura de una memoria consiste en un arreglo lineal de las diversas locaciones ordenadas en forma consecutiva, tal como se muestra en la figura 4.3. Cada una de las direcciones identificadas en la figura con un valor numérico se corresponde con una palabra específica almacenada en la misma. (En este ejemplo, la palabra se compone de cuatro bytes.) El número que identifica en forma unívoca cada palabra se define como su **dirección**. Dado que las direcciones se cuentan en secuencia, a partir de cero, la dirección más alta corresponde a una unidad menos que el tamaño de la memoria. La última dirección de una memoria de 2^{32} bytes es $2^{32} - 1$, en tanto que la primera dirección es 0.

El modelo de memoria a ser utilizado durante el resto del capítulo es el que se ilustra en la figura 4.4. La memoria tiene un **espacio de direcciones** de 32 bits, lo que significa que el programador puede acceder a un byte de memoria ubicado en cualquier posición en el rango de 0 a $2^{32} - 1$. El espacio de direcciones de la arquitectura de este ejemplo está dividido en diferentes sectores, los que se utilizan para el sistema operativo, para los elementos de entrada-salida, para los programas del usuario y para la pila del sistema. La distribución de estos sectores forma el **mapa de memoria** que se ilustra en la figura 4.4. El mapa de memoria puede diferir entre una implementación y otra, y esta es una de las

causas por las cuales algunos programas compilados para el mismo tipo de procesador pueden no ser compatibles entre sistemas.

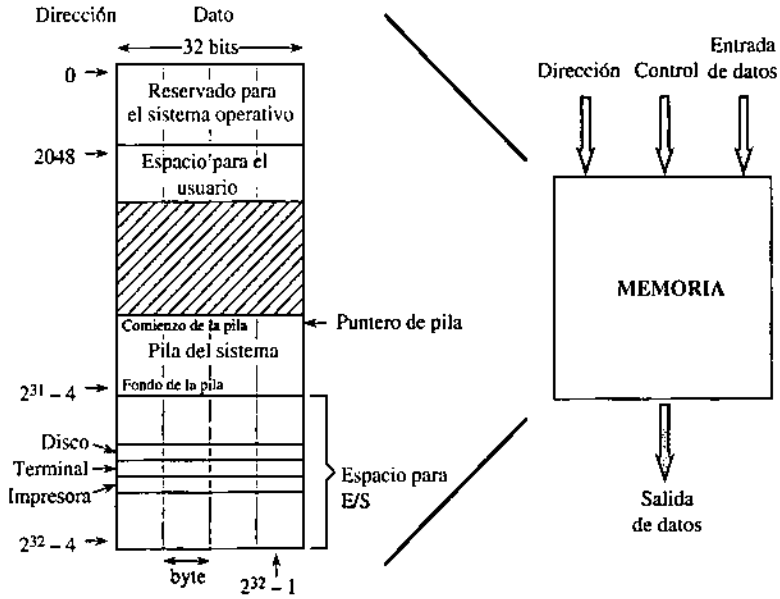


Figura 4.4 • Un mapa de memoria para un ejemplo de arquitectura. (No dibujado en escala.)

Las $2^{11} = 2048$ primeras direcciones del mapa de memoria se reservan para el sistema operativo. El espacio para el usuario es aquel que se reserva para la carga de los programas ensamblados del usuario, y puede crecer, durante la operación, desde la dirección 2048 hasta que se encuentre con la pila del sistema. La pila del sistema comienza en la dirección $2^{31} - 4$ y crece hacia las direcciones inferiores. La zona del espacio de direcciones ubicada entre 2^{31} y $2^{32} - 1$ se reserva para los dispositivos de entrada-salida. En consecuencia, el mapa de memoria no está compuesto enteramente por memoria real, y de hecho pueden existir espacios en los que no haya ni memoria real ni direcciones de dispositivos de entrada-salida. Dado que los dispositivos de entrada-salida se tratan como posiciones de memoria, la lectura y escritura de dichos dispositivos de entrada-salida se puede realizar a través de los mismos comandos de lectura y escritura utilizados para la memoria del sistema. En estas circunstancias, se habla de un sistema de **entrada-salida mapeado en memoria**.

Es importante que quede clara la diferenciación entre lo que significa una dirección y lo que significa un dato. En este ejemplo, la palabra binaria que representa una dirección de memoria tiene 32 bits, en tanto que una palabra de datos también tiene 32 bits, a pesar de lo cual no son la misma cosa. Una dirección es un puntero a una posición de memoria, la cual contiene un dato.

En este capítulo se supone que la memoria de la computadora está organizada en un único espacio de direcciones. La expresión **espacio de direcciones** se refiere al rango numérico de direcciones de memoria al que puede hacer referencia la unidad de proce-

so. En el capítulo 7 ("Memorias") se verá que existen otras formas de organización de la memoria, pero, por el momento, se supondrá que la memoria, tal como se la ve desde la CPU, tiene un único rango de direcciones. Lo que decide el tamaño de ese rango de direcciones es el tamaño de la dirección de memoria que la CPU puede colocar sobre el bus de direcciones durante las operaciones de lectura y escritura. Una palabra de dirección formada por n bits puede especificar una de 2^n direcciones. Esta memoria puede plantearse como una memoria con un espacio de direcciones de n bits, o, en forma equivalente, como una memoria con un espacio de direcciones de 2^n bytes. Por ejemplo, una memoria con un espacio de direcciones de 32 bits tiene una capacidad máxima de 2^{32} bytes (4 Gbytes). El rango de las direcciones de memoria va desde 0 hasta $2^{32} - 1$, o lo que es igual, desde 0 hasta 4.294.967.295 si se lo expresa en decimal, o bien, expresado en el sistema hexadecimal, más sencillo de manejar, desde 00000000H hasta FFFFFFFFH. (En la mayoría de los lenguajes simbólicos, la H indica un número hexadecimal.)

4.1.3 La unidad central de proceso (CPU)

Una vez familiarizados con los componentes básicos del bus del sistema y de la memoria, corresponde explorar la estructura interna de la CPU. Como mínimo, la estructura de la CPU consiste en una **sección de datos**, formada por los registros y la unidad aritmético-lógica (ALU), y una **sección de control**, la que interpreta las instrucciones y realiza las trasferencias entre registros, como se observa en la figura 4.5. La sección correspondiente a los datos se conoce también como **trayecto de datos** (*datapath*).

La unidad de control es la responsable de la ejecución de las instrucciones del programa, las que se almacenan en la memoria principal. (En el análisis que sigue se supondrá que la unidad de control ejecuta el código de máquina de a una instrucción por vez, aun cuando en el capítulo 9 se planteará la arquitectura de muchos procesadores modernos que pueden procesar varias instrucciones en forma simultánea.) Existen dos registros que forman la interfaz entre la unidad de control y la unidad de datos, llamados **contador de programa**[1] (PC, *program counter*) y **registro de instrucción** (IR, *instruction register*). El contador de programa contiene la dirección de la instrucción en ejecución. La instrucción a la que apunta el PC se rescata de memoria y se almacena en el IR, desde donde se la interpreta. Los pasos que lleva a cabo la unidad de control en la ejecución de un programa son los que se detallan:

1. Búsqueda en memoria de la próxima instrucción a ser ejecutada.
2. Decodificación del código de operación.
3. Búsqueda de operandos en memoria, si los hubiera.

1. En los procesadores Intel, el contador de programa se conoce como puntero a instrucciones (*instruction pointer*, IP).

4. Ejecución de la instrucción y almacenamiento de los resultados.
5. Vuelta al paso 1.

Este proceso se conoce como **ciclo de búsqueda-ejecución**.

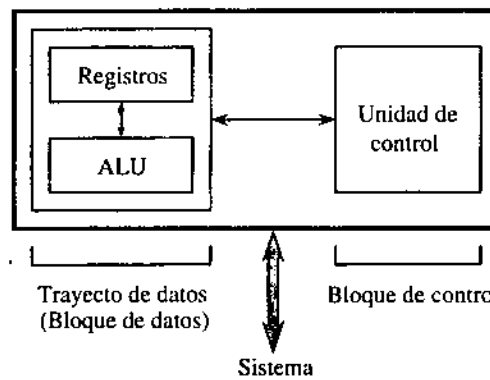


Figura 4.5 • La CPU vista en alto nivel.

La unidad de control es la responsable de coordinar las distintas unidades que intervienen en la ejecución de un programa de computadora. Puede pensársela como "una computadora dentro de la computadora", entendiendo que toma decisiones acerca del comportamiento del resto de la máquina. La unidad de **control** se trata en detalle en el capítulo 6.

El trayecto de datos está constituido por un **conjunto** o **bloque de registros** y por la unidad aritmético-lógica, tal como se muestra en la figura 4.6. La figura ilustra el camino de datos de un modelo de procesador que será utilizado en el resto del capítulo.

El conjunto de registros de la figura puede verse como una memoria, pequeña y rápida, separada de la memoria del sistema, que se utiliza para el almacenamiento temporario durante las operaciones de cálculo. Los tamaños típicos del conjunto de registros van desde algunos pocos a algunos miles de registros. Al igual que la memoria principal del sistema, cada registro del conjunto recibe una dirección ordenada secuencialmente a partir de cero. Estas direcciones de registros son mucho más chicas que las direcciones de memoria: un conjunto de registros que contenga 32 registros requeriría una palabra de direcciones de solo cinco bits. Las diferencias principales entre el conjunto de registros y la memoria del sistema derivan de que los registros están contenidos dentro de la CPU; por consiguiente, funcionan con mayor rapidez que aquella. Una instrucción que se ejecuta sobre operandos almacenados en el conjunto de registros puede ejecutarse unas diez veces más rápido que si los operandos estuviesen en memoria. Por esta razón, los programas que hacen uso intensivo de los registros de la CPU son más rápidos que los programas equivalentes que hacen uso intensivo de la memoria, aunque, para realizar las mismas tareas, hiciesen falta más operaciones cuando los operandos se encuentran almacenados en registros que si estuviesen almacenados en memoria.

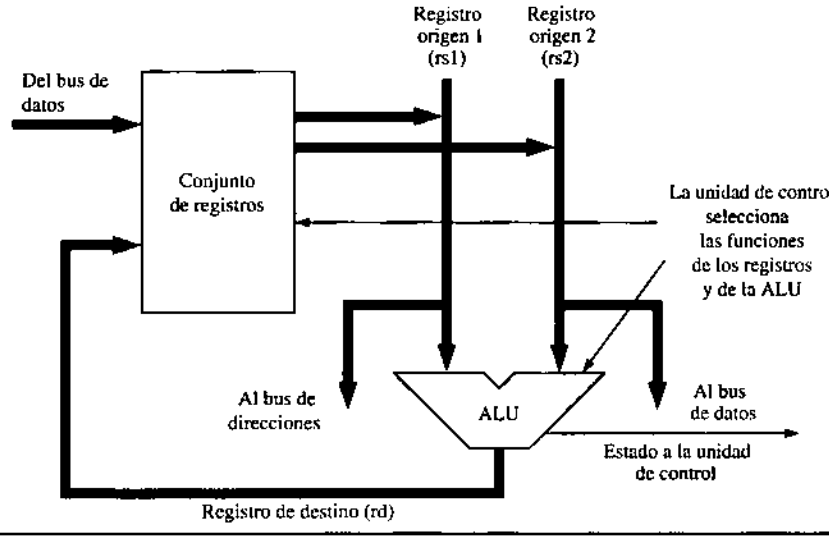


Figura 4.6 • Ejemplo de un trayecto de datos.

Nótese que dentro del trayecto de datos de la figura 4.6 existen varios buses. Tres buses conectan el trayecto de datos con el bus del sistema. Esto permite la trasferencia de los datos entre la memoria principal y el conjunto de registros. Tres buses adicionales conectan el conjunto de registros con la unidad aritmético-lógica. Estos buses permiten la búsqueda simultánea de dos operandos almacenados en el conjunto de registros, para ser procesados por la ALU, tras lo cual se devuelve el resultado en el conjunto de registros.

La ALU implementa una variedad de operaciones de uno y dos operandos, entre las cuales se incluyen la suma, el producto y las operaciones lógicas de disyunción (`or`), conjunción (`and`) y negación (`not`). La unidad de control selecciona las operaciones y los operandos a utilizar por aquellos. Los dos operandos se trasfieren desde el bloque de registros a través de dos buses que se conectan con los registros de origen 1 (`rs1`) y 2 (`rs2`). La salida de la ALU se coloca en el bus que conduce al registro destino (`rd`) con el objeto de devolver el resultado al registro correspondiente dentro del conjunto de registros. En la mayoría de los sistemas, estas conexiones también incluyen la conexión con el bus del sistema de modo de poder acceder a la memoria y a los dispositivos de entrada-salida. En la figura, estas conexiones son las rotuladas como "del bus de datos", "al bus de datos" y "al bus de direcciones".

El conjunto de instrucciones

El **conjunto de instrucciones** es la colección de instrucciones que un procesador puede ejecutar, lo que, en efecto, define al procesador. Los conjuntos de instrucciones pueden ser completamente diferentes de un procesador a otro. Difieren en los tamaños de las instrucciones, el tipo de operaciones que permiten, el tipo de operandos sobre los que pueden operar y los tipos de resultados que pueden entregar. Esta incompatibilidad en los conjuntos de instrucciones contrasta claramente con la compatibilidad de los lenguajes

de alto nivel, como C, Pascal y Ada. Los programas escritos en estos lenguajes de alto nivel pueden ejecutarse casi sin cambios en diferentes procesadores, siempre y cuando sean **recompilados** para el procesador a utilizar.

Una excepción a esta incompatibilidad entre los lenguajes de máquina se puede ver en los programas compilados en código Java, dado que es un lenguaje de máquina para una **máquina virtual**. Estos se ejecutarán sin cambios en cualquier procesador que esté ejecutando la máquina virtual de Java. Esta máquina virtual, escrita en el lenguaje simbólico del procesador en que se utiliza, intercepta cada uno de los bytes de código Java y lo ejecuta como si estuviese funcionando sobre una máquina "real" para Java. El caso de estudio del final del capítulo permite ver más detalles del tema.

Debido a la incompatibilidad planteada entre los conjuntos de instrucciones, los sistemas de computación se definen, a menudo, por el tipo de CPU que incluyen. El conjunto de instrucciones determina cuáles programas puede ejecutar el sistema y tiene un impacto importante sobre el rendimiento. Los programas compilados para una computadora personal IBM (o compatible) utilizan el conjunto de instrucciones de una CPU 80x86, donde la "x" se reemplaza por un dígito que corresponde a la versión, tal como 80586, procesador normalmente conocido como Pentium. Estos programas no funcionarán sobre una computadora Apple Macintosh o sobre una IBM RS6000, dado que estas dos máquinas ejecutan el juego de instrucciones del procesador **PowerPC** de Motorola. Esto no significa que todos los sistemas de computación que utilizan la misma CPU pueden ejecutar los mismos programas. Un programa PowerPC escrito para IBM RS6000 no funcionará en la Macintosh sin grandes modificaciones, debido a las diferencias entre los sistemas operativos y las convenciones de entrada-salida.

En secciones posteriores de este capítulo se analizará en detalle un conjunto de instrucciones específico.

Software para la generación de programas en lenguaje de máquina

Un **compilador** es un programa de computadora que transforma programas escritos en un lenguaje de alto nivel, como C, Pascal o Fortran, en lenguaje de máquina. Los compiladores para un mismo lenguaje de alto nivel generalmente presentan la misma "cara", definida como la parte que reconoce las sentencias del lenguaje de alto nivel. No obstante, sus estructuras tendrán diferentes aspectos en lo que hace a su salida hacia el procesador para el que se produce la compilación. Esta salida del compilador es la parte del programa responsable de la generación de código de máquina para un procesador específico. Por otra parte, un mismo programa C, compilado por distintos compiladores C para la misma máquina, puede generar diferentes compilaciones aun cuando el código fuente sea el mismo, como se verá posteriormente.

En el proceso de compilación de un programa (normalmente conocido como **proceso de traducción**), el programa fuente escrito en lenguaje de alto nivel se transforma en **lenguaje simbólico** (*assembly language*), luego de lo cual un programa **ensamblador** (*assembler*) traduce este lenguaje simbólico hacia el código de máquina del procesador

de destino. Estas traducciones se producen, respectivamente, en los llamados **momento de la compilación** (*compile time*) y **momento del ensamble** (*assembly time*). El programa objeto resultante puede vincularse con otros programas objeto en el **momento del enlace** (*link time*). El programa objeto, normalmente almacenado en un disco, se carga en la memoria principal en el **momento de la carga** (*load time*) y se ejecuta desde la CPU en el **momento de la ejecución** (*run time*).

Si bien la mayor parte del código se escribe en lenguajes de alto nivel, es posible el uso del lenguaje simbólico para escribir programas que pueden ser críticos en espacio o tiempo. Por otra parte, no siempre se consiguen compiladores para algunos procesadores dedicados a aplicaciones especiales, o bien, sus compiladores pueden no ser aptos para expresar operaciones especiales. En estos casos también puede surgir la necesidad de la programación en lenguaje simbólico.

Los lenguajes de alto nivel permiten ignorar, durante la codificación, la arquitectura de la computadora en que se va a ejecutar el programa. Al nivel del lenguaje de máquina, no obstante, la arquitectura subyacente se vuelve un elemento de consideración principal. Un programa escrito en un lenguaje de alto nivel, como C, Pascal o Fortran, puede tener el mismo aspecto y ejecutarse correctamente sobre distintos sistemas luego de la compilación. El código objeto producido por el compilador para cada máquina será totalmente diferente para cada sistema, aunque los sistemas utilicen el mismo conjunto de instrucciones. Tal es el caso de aquellos programas compilados para PowerPC a ser ejecutados en Macintosh contra los que se ejecutan sobre una RS6000 IBM.

Habiendo analizado las características de la CPU, la memoria y el bus del sistema, se procederá ahora a analizar en detalle un conjunto de instrucciones a ser utilizado como modelo, llamado ARC.

4.2 ARC, una computadora RISC

En el resto del capítulo se analizará un modelo de arquitectura basado en el procesador **SPARC** (*Scalable Processor Architecture*) desarrollado por Sun Microsystems a mediados de los años ochenta. El procesador SPARC se convirtió en una arquitectura popular desde el mismo instante de su introducción, lo que se justifica en parte debido a su naturaleza abierta: la definición completa de la arquitectura SPARC se divulga al poco tiempo de su aparición. En este capítulo se analiza un subconjunto de SPARC, al que se denominará **ARC** (*A RISC Computer*). RISC es también un acrónimo para la definición de una **computadora con un conjunto reducido de instrucciones** (RISC, *Reduced Instruction Set Computer*), la cual se analiza en el capítulo 9. La configuración ARC contiene la mayoría de las características importantes de la arquitectura SPARC, pero sin algunas de las prestaciones más complejas presentes en un procesador comercial.

4.2.1 Memoria en ARC

ARC es una máquina de 32 bits con capacidad de direccionamiento de memoria por bytes; puede manejar tipos de datos de 32 bits, pero toda la información se almacena en memoria en la forma de bytes. La dirección de la palabra de 32 bits es la del byte almacenado en la menor de las direcciones de la palabra. Como se ha descripto anteriormente en el contexto de la figura 4.4, ARC tiene un espacio de memoria de 32 bits, que en la arquitectura de este ejemplo se divide en distintas regiones a ser usadas por el código del sistema operativo, el código del programa del usuario, la pila del sistema (utilizada para el almacenamiento temporario de información) y la entrada y salida. Estas regiones de memoria se detallan a continuación:

- Las primeras 2048 direcciones del mapa de memoria se reservan para ser usadas por el sistema operativo.
- El espacio del usuario es el que se reserva para la carga de los programas ensamblados por el usuario. Puede crecer durante el funcionamiento desde la dirección 2048 hasta que se encuentre con la pila del sistema.
- La pila del sistema se inicia en la dirección $2^{31} - 4$ y crece hacia direcciones inferiores de memoria. La causa por la cual se organiza el crecimiento de los programas en el sentido de memoria creciente y la pila en sentido decreciente puede verse en la figura 4.4; esta organización acepta tanto una combinación de programas grandes con pilas pequeñas como la de programas pequeños y pilas grandes.
- La zona del mapa de direcciones ubicada entre 2^{31} y $2^{32} - 1$ se reserva para los dispositivos de entrada-salida. Cada dispositivo tiene asignada una serie de direcciones de memoria para el almacenamiento de sus datos, en lo que se denomina "entrada-salida mapeada en memoria".

ARC tiene diferentes tipos de datos (byte, media palabra, entero, etc.), pero, por el momento, se considerará solo el tipo de datos entero de 32 bits. Cada entero se almacena en memoria como un conjunto de cuatro bytes. La arquitectura ARC funciona con el byte más significativo almacenado en la menor de las direcciones asignada a la palabra, lo que se conoce como almacenamiento ***big endian***. La máxima dirección disponible para el almacenamiento de un byte en ARC es $2^{32} - 1$, de modo que la dirección de la palabra más alta de memoria está ubicada tres bytes por debajo de esta, en $2^{32} - 4$.

4.2.2 El conjunto de instrucciones ARC

Con el objeto de obtener detalles del conjunto de instrucciones ARC, se hace necesario revisar primero las características de la CPU:

- ARC tiene 32 registros de uso general, de 32 bits cada uno, así como un contador de programa (PC) y un registro de instrucción (IR).
- Existe un **registro de estado del procesador** (**PSR**, *processor status register*) que contiene información acerca del estado del procesador, incluida la información referida a los resultados de las operaciones aritméticas. Las "banderas aritméticas" del registro de estado se denominan **códigos de condición**. Especifican si en una operación aritmética determinada el resultado dio cero (z), si dio un número negativo (n), si se obtuvo un arrastre a la salida de la unidad aritmético-lógica de 32 bits (c), o si se produjo un desborde (v). El bit v se activa cuando el resultado de la operación aritmética es demasiado grande como para ser manejado por la unidad aritmético-lógica.
- El tamaño de todas las instrucciones es de una palabra (32 bits).
- ARC es una máquina de arquitectura **carga-descarga** (*load-store*). Las únicas instrucciones permitidas para el acceso a memoria cargan un valor hacia alguno de los registros, o almacenan en una posición de memoria el contenido de alguno de los registros. Todas las operaciones aritméticas se ejecutan sobre operandos contenidos en registros y los resultados también quedan almacenados en un registro. El conjunto de instrucciones del procesador SPARC, sobre el que se basa ARC, tiene aproximadamente 200 instrucciones. La figura 4.7 ilustra un subconjunto de 15 de esas instrucciones. Cada instrucción está representada por un código **nemotécnico**, definido como un nombre utilizado para simbolizar la instrucción.

	Mnemónico	Función
Memoria	ld	Cargar un registro desde memoria
	st	Salvar un registro en memoria
Lógicas	sethi	Cargar los 22 bits más significativos de un registro
	andcc	Producto lógico bit a bit (Y)
	orcc	Suma lógica bit a bit (O)
	orncc	Suma lógica negada bit a bit (NOR)
	srl	Desplazar a derecha (lógico)
Aritméticas	addcc	Sumar
Control	call	Salto (llamado) a subrutina
	jmpl	Retorno de subrutina
	be	Bifurcación (salto por igual)
	bneg	Bifurcación (salto) por negativo
	bcs	Bifurcación (salto) por arrastre
	bvs	Bifurcación (salto) por desborde
	ba	Bifurcación (salto) incondicional

Figura 4.7 • Subconjunto de instrucciones para la arquitectura de programación ARC.

Instrucciones para el movimiento de datos

Las primeras dos instrucciones, **ld** (*load* [cargar]) y **st** (*store* [descargar o almacenar]) transfieren una palabra entre la memoria principal y alguno de los registros de ARC. Estas son las únicas instrucciones que permiten el acceso a memoria dentro de ARC.

La instrucción **sethi** carga los 22 bits más significativos de un registro con la constante de 22 bits contenida dentro de la instrucción. Se la utiliza habitualmente para construir una constante arbitraria de 32 bits a ser almacenada en un registro, en conjunto con otra instrucción que carga los 10 bits menos significativos del registro.

Instrucciones aritméticas y lógicas

Las instrucciones **andcc**, **orcc** y **orncc** realizan operaciones Y, O y NOR, respectivamente, sobre sus operandos. Uno de los dos operandos origen debe estar en un registro. El otro puede estar en un registro o puede ser una constante incluida en la instrucción, expresada en 13 bits en notación de complemento a dos, la que se extiende a 32 bits cuando se la utiliza. El resultado se almacena en un registro.

Para el caso de la instrucción `andcc`, cada bit del resultado se coloca en 1 si los bits correspondientes de ambos operandos son simultáneamente 1; en caso contrario, el bit del resultado vale 0. En la operación `orcc`, cada bit del registro es 1 si al menos uno de los correspondientes bits de los operandos vale 1; en caso contrario, el bit correspondiente del resultado vale 0. La operación `orncc` es complementaria de `orcc`, de modo tal que cada bit del resultado vale 0 si uno o los dos bits correspondientes en los operandos vale 1; en caso contrario, el bit del resultado se coloca en 1. Los sufijos "`cc`" especifican que luego de realizada la operación deben actualizarse los códigos de condición del registro de estado de modo que permitan reflejar el resultado de la operación realizada. En particular, el bit `z` se coloca en 1 si todos los bits del registro que contiene el resultado son cero, el bit `n` se pone en 1 si el bit más significativo del resultado es 1, y los bits `c` y `v`, en el caso de estas instrucciones en particular, se colocan en 0. (¿Por qué?)

Las instrucciones de desplazamiento permiten desplazar el contenido de un registro hacia otro. La instrucción **srl** (*shift right logical* [desplazamiento lógico a derecha]) desplaza un registro hacia la derecha y carga ceros en el (los) bit(s) más significativo(s). La instrucción **sra** (*shift right arithmetic* [desplazamiento aritmético a derecha]), que no se indica en la tabla, desplaza el contenido original del registro a la derecha y almacena una copia del bit más significativo del contenido original en los bits vacíos creados por el desplazamiento del registro. Esta operación da por resultado la extensión del signo del operando, preservando así su signo aritmético.

La instrucción **addcc** realiza sobre sus operandos una suma de 32 bits en complemento a dos.

Instrucciones de control

Las instrucciones **`call`** y **`jmpl`** forman un par utilizado para el llamado de subrutinas y el retorno desde las mismas, respectivamente. La instrucción `jmpl` también se utiliza para transferir el control a otro sector del programa.

Las últimas cinco instrucciones son instrucciones de **salto o bifurcación condicional**. Las instrucciones **`be`**, **`bneg`**, **`bcs`**, **`bvs`** y **`ba`** provocan un salto en la secuencia de ejecución de un programa. Se las llama condicionales debido a que verifican uno o más de los códigos de condición del registro de estado, y provocan el salto si los bits indican que la condición verificada se cumple. Se utilizan para la implementación de construcciones de alto nivel, tales como `goto`, `if-then-else` y `do-while`. Las secciones siguientes describen estas instrucciones en forma detallada y ofrecen ejemplos de su utilización.

4.2.3 Formato del lenguaje simbólico de ARC

Cada lenguaje simbólico tiene su propia sintaxis. En este caso, la sintaxis a respetar es la del lenguaje de programación simbólico de SPARC, que se muestra en la figura 4.8. El formato incluye cuatro campos: un campo optativo para los rótulos, un campo para el código de operación, uno o más campos para especificar los operandos origen y destino (si existieran operandos), y un último campo optativo para los comentarios. Los rótulos se forman con cualquier combinación de los caracteres alfabéticos o numéricos y con los símbolos correspondientes al guión bajo (_), al signo monetario ($) y al punto (.), siempre y cuando el primer carácter no sea un dígito. Todo rótulo debe terminar en dos puntos (:). El lenguaje es sensible al tipo de letra, por lo que hace distinción entre letras mayúsculas y minúsculas. El lenguaje ofrece "formato libre", lo que significa que cualquier campo puede empezar en cualquier columna, debiendo mantenerse el ordenamiento relativo de izquierda a derecha.

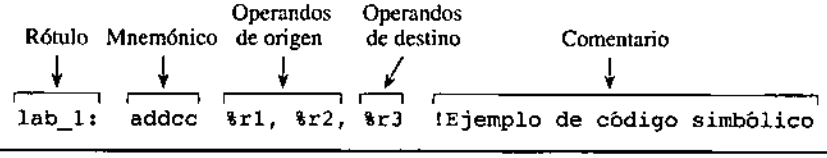


Figura 4.8 • Formato de una sentencia en el lenguaje simbólico SPARC (también ARC).

La arquitectura ARC contiene 32 registros denominados `%r0 - %r31`, cada uno de los cuales contiene una palabra de 32 bits. También existe un registro de estado del procesador (PSR), que describe el estado vigente del procesador, y un **contador de programa** (PC) de 32 bits, que controla la dirección de la instrucción en ejecución, según se ilustra en la figura 4.9. El registro de estado del procesador se denomina `%psr`, en tanto que el contador de programa se designa como `%pc`. El registro `%r0` siempre contiene el valor

0, el cual no puede modificarse. Los registros `%r14` y `%r15` tienen aplicación adicional como **puntero de pila** (`%sp`, *stack pointer*) y **registro de enlace** (*link register*), respectivamente, según se describe más adelante.

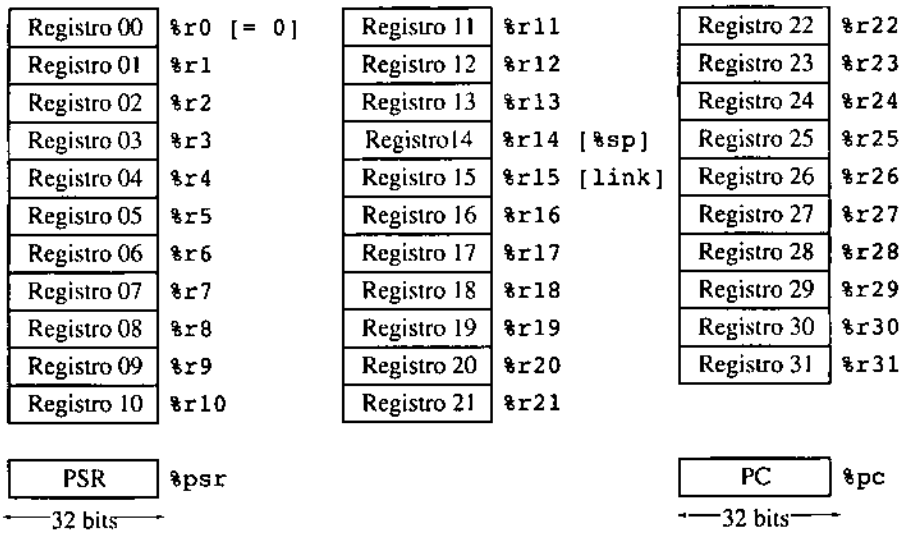


Figura 4.9 • Registros en ARC accesibles al programador.

Los operandos que aparecen en una sentencia del lenguaje simbólico se separan por comas, y el operando de destino siempre debe aparecer, dentro del campo de operandos, en la posición de la derecha. Por consiguiente, el ejemplo de la figura 4.8 especifica la suma de los registros `%r1` y `%r2`, y el almacenamiento del resultado en `%r3`. Si en el campo de operandos, en la posición de destino, aparece el registro `%r0`, el resultado se descarta. Los operandos numéricos se representan, por defecto, en el sistema de numeración decimal, por lo que la sentencia

```
addcc  %r1, 12, %r3
```

muestra una operación de suma entre el contenido del registro `%r1` y un operando de valor $(12)_{10}$, cuyo resultado se almacenará en el registro `%r3`. Los números se interpretan en el sistema decimal, a menos que se los preceda por "`0x`" o se los termine en "`H`", cualquiera de los cuales indica que el número es hexadecimal. El campo de comentarios se encuentra a continuación del campo de operandos: se inicia con un signo de admiración y termina al final del renglón.

4.2.4 Formatos de instrucción en ARC

El **formato de instrucción** define la manera en que el ensamblador distribuye los diversos campos de una instrucción y la forma en que los interpreta la unidad de control de ARC. La arquitectura ARC tiene unos pocos formatos de instrucción. Los cinco forma-

tos son: **SETHI**, **Salto**, **Llamada** (*CALL*), **Aritmético** y **Memoria**, de acuerdo con lo que se describe en la figura 4.10. Cada instrucción tiene una representación nemotécnica, tal como "`ld`", y un código de operación. Un formato particular de instrucción puede tener más de un campo de operandos, los que en forma colectiva identifican una instrucción en una de sus diversas formas. (Nótese que los formatos de instrucción mencionados no se corresponden directamente con la clasificación de las instrucciones en cuatro grupos, planteada en la figura 4.7).

Los dos bits más significativos (a la izquierda de la palabra) forman el campo `op` correspondiente al código de operación, el que identifica el formato. Los dos formatos correspondientes a SETHI y a las instrucciones de salto contienen `00` en el campo `op`, por lo cual pueden juntarse para considerar un único formato. Quien determina si el formato corresponde a SETHI o a un salto es el dato contenido en el campo `op2` (`010` = salto, `100` = SETHI). En el formato de salto, el bit 29 siempre vale cero. El campo de cinco bits `rd` identifica el registro al que se aplicará la instrucción SETHI.

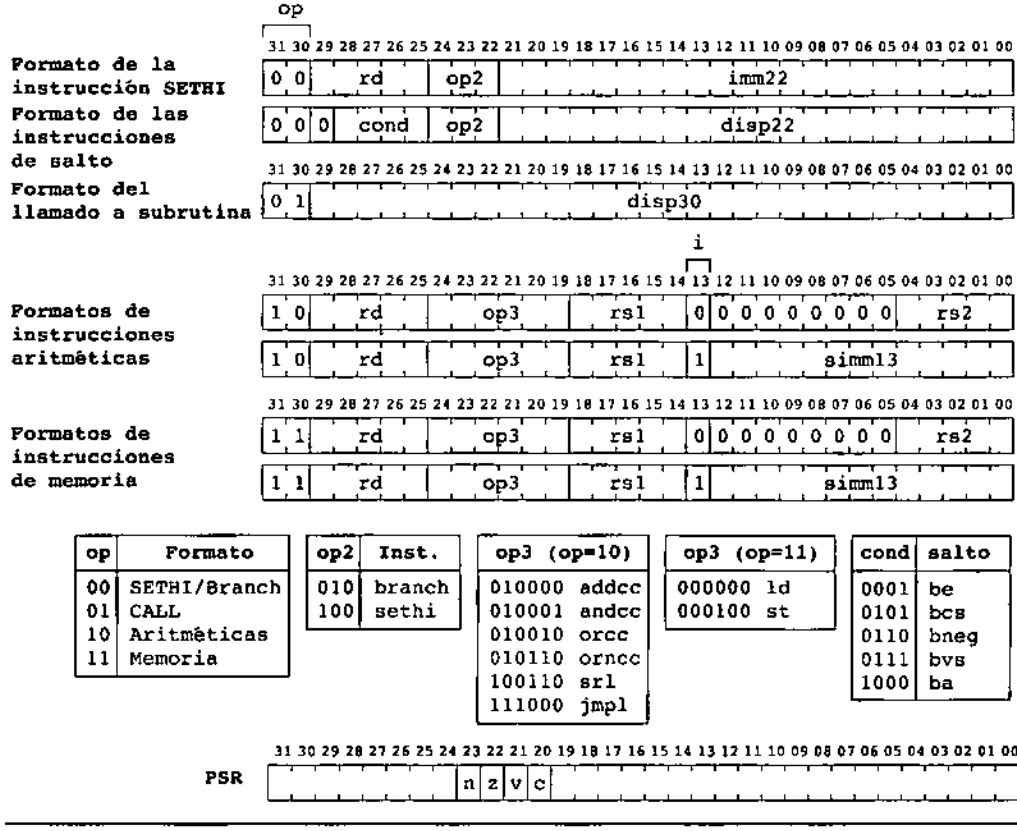


Figura 4.10 • Formatos de instrucción y contenido del PSR en ARC.

El campo `cond` identifica el tipo de salto condicional, que se basa en los códigos de condición (`n`, `z`, `v` y `c`) del registro de estado PSR, de acuerdo con lo que puede observarse en la parte inferior de la figura 4.10. El resultado de la ejecución de cualquier instrucción cuyo mnemónico termine en "`cc`" acomoda los bits de código de condición de modo que `n = 1` si el resultado de la operación es negativo; `z = 1` si el resultado es cero; `v = 1` si la operación causa un desborde; y `c = 1` si la operación genera un arrastre. Las instrucciones que no terminan en "`cc`" no afectan los códigos de condición. Cada uno de los campos `imm22` y `disp22` contiene una constante de 22 bits que se utiliza como operando en el formato SETHI (para `imm22`) o para el cálculo del desplazamiento que permite determinar una dirección de salto (en el caso de `disp22`).

El formato correspondiente a una llamada tiene solo dos campos: el campo `op`, que contiene la palabra binaria `01`, y un campo `disp30`, que contiene un desplazamiento de 30 bits que se utiliza para determinar la dirección de la rutina invocada.

Los formatos aritmético (`op = 10`) y memoria (`op = 11`) utilizan campos `rd` para identificar un registro origen en el caso de `st`, o un registro de destino para las instrucciones restantes. El campo `rs1` identifica el primer registro origen y el campo `rs2` identifica el segundo registro origen. El campo `op3` correspondiente al código de operación identifica la instrucción, de acuerdo con las tablas `op3` indicadas en la figura 4.10.

El campo `simm13` es un valor inmediato de 13 bits, que se extiende con signo a 32 bits para el segundo registro origen cuando el campo `i` (inmediato) vale `1`. El concepto de "extender con signo" significa que el bit de signo (el más significativo) se copia en las posiciones restantes de una palabra de 32 bits, antes de ser sumado a `rs1` en este caso. Esto asegura que un número negativo en complemento a dos sigue siendo negativo tras la conversión a 32 bits (así como un número positivo continúa siendo positivo). Por ejemplo, $(-13)_{10} = (1\ 1111\ 1111\ 0011)_2$, y luego de la extensión a un formato entero de 32 bits se tiene $(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0011)_2$, que sigue siendo equivalente a $(-13)_{10}$.

Las instrucciones aritméticas necesitan dos operandos origen y un operando destino, lo que hace un total de tres operandos. Las instrucciones de memoria solo necesitan dos operandos, uno para la dirección y uno para el dato. El campo origen remanente, sin embargo, se utiliza también para la dirección. Cuando `i = 0`, los operandos contenidos en los campos `rs1` y `rs2` se suman para obtener la dirección. Cuando `i = 1`, la dirección se obtiene sumando los campos `rs1` y `simm13`. Para los primeros ejemplos a plantear, se utilizará `%r0` para `rs2`, con lo cual solo se especificará el operando origen restante.

4.2.5 Formatos de datos en ARC

ARC soporta 12 formatos de datos diferentes, según se ilustra en la figura 4.11. Los formatos de datos se agrupan en tres tipos: entero signado, entero sin signo y punto flotante. Dentro de estos tipos, los tamaños admisibles son byte (8 bits), media palabra (16

bits), palabra (32 bits), palabra **rotulada** (32 bits, de los cuales los dos bits menos significativos forman un **rótulo** o referencia, en tanto que los restantes 30 bits forman el valor), **palabra doble** (64 bits) y **palabra cuádruple** (128 bits).

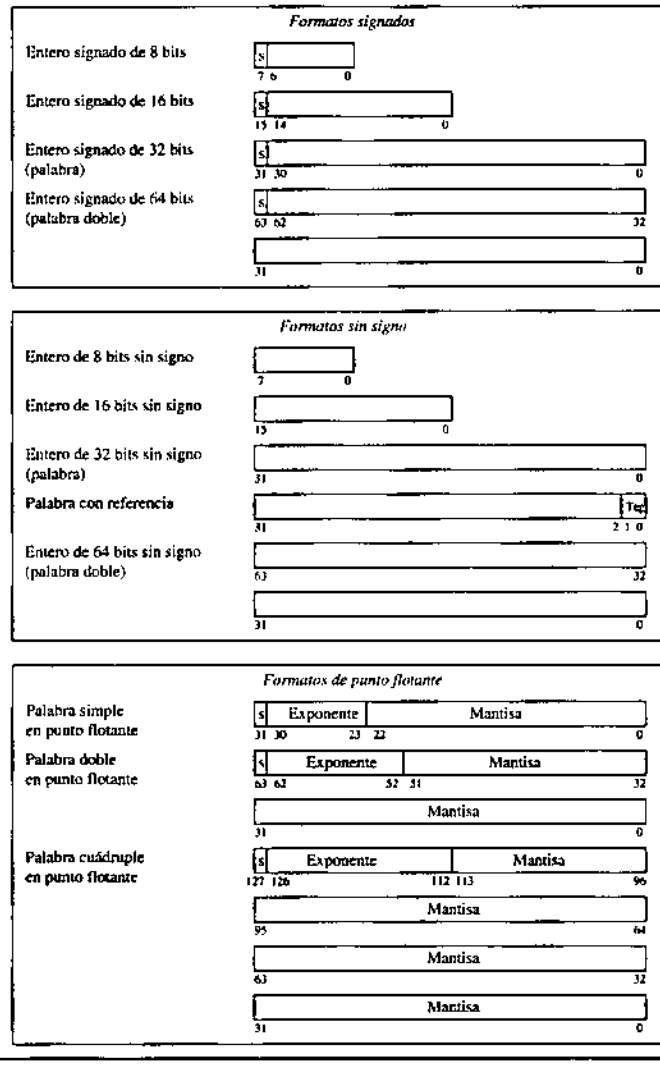


Figura 4.11 • Formatos de datos en ARC.

En realidad, ARC no distingue entre enteros signados o no signados. Ambos se manejan y almacenan como enteros representados en notación de complemento a dos. Varía solo su interpretación. En particular, un subconjunto de las instrucciones de salto supone que los valores que se comparan son valores con signo, en tanto que el otro subconjunto supone que no tienen signo. En forma similar, el bit c indica un desborde en las operaciones enteras sin signo, en tanto que el bit v lo identifica para operaciones con signo.

La palabra rotulada utiliza los dos bits menos significativos para indicar un **desbor-**

de, en el caso en que se intente almacenar un valor mayor que 30 bits en los 30 bits asignados en la palabra. Se utiliza en lenguajes con datos dinámicos, como Lisp y Smalltalk. En forma genérica, un 1 en cualquier bit del campo de referencia indica una situación de desborde en esa palabra. Las referencias pueden utilizarse para asegurar que se cumplan las condiciones de alineación requeridas (que una palabra empiece en una dirección múltiplo de cuatro bytes, que la palabra doble empiece en un múltiplo de ocho bytes, etc.), especialmente en el caso de los punteros.

Los formatos de punto flotante se establecen de acuerdo con la norma IEEE 754-1985 (véase el capítulo 2). Existen instrucciones especiales que invocan formatos de punto flotante que no se describen aquí y que pueden encontrarse en la literatura de SPARC.

4.2.6 Descripción de las instrucciones de ARC

Siendo ya conocidos los formatos de instrucción, se puede desarrollar una descripción detallada de las 15 instrucciones de la figura 4.7. Esta sección se ocupa de esa descripción. La traducción a código objeto se agrega solo como referencia, ya que se describe en detalle en el próximo capítulo. En las descripciones, la referencia al contenido de una posición de memoria (para `ld` y `st`) se indica con corchetes, tal como en "`ld [x], %r1`", que copia el contenido de la dirección `x` en el registro `%r1`. La referencia a la dirección de una posición de memoria se especifica directamente, sin corchetes, como ocurre en "`call sub_r`", instrucción que genera un llamado a la subrutina `sub_r`. Solo `ld` y `st` pueden acceder a memoria, razón por la cual admiten el uso de corchetes. Los registros siempre se mencionan en función de su contenido, nunca en términos de una dirección; por consiguiente, no hay necesidad alguna de encerrar entre corchetes las referencias a los registros.

Instrucción: `ld`

Descripción: Carga un registro desde la memoria principal. La dirección de memoria debe estar alineada con una frontera de palabra (lo que significa que la dirección debe ser múltiplo de 4). Se calcula la dirección sumando el contenido del registro del campo `rs1` con el contenido del registro indicado en el campo `rs2` o con el valor del campo `simm13`, según corresponda al contexto.

Ejemplo de uso: `ld [x], %r1`
o bien `ld [x], %r0, %r1`
o `ld %r0+x, %r1`

Significado: Copiar el contenido de la dirección `x` de memoria en el registro `%r1`.

Código objeto: `1100 0010 0000 0000 0010 1000 0001 0000` `(x = 2064)`

Instrucción: `st`

Descripción: Almacena el contenido de un registro en la memoria principal. La dirección de memoria debe estar en frontera de palabra. La dirección se determina sumando el contenido del registro indicado en el campo `rs1` con el contenido del registro indicado en el campo `rs2` o con el valor del campo `simm13`, según corresponda. El campo `rd` de esta instrucción se utiliza para el registro origen.

Ejemplo de uso: `st %r1, [x]`

Significado: Copiar el contenido el registro `%r1` en la posición de memoria x.

Código objeto: `1100 0010 0010 0000 0010 1000 0001 0000` `(x = 2064)`

Instrucción: `sethi`

Descripción: Carga los 22 bits más significativos de un registro y coloca en cero sus diez bits menos significativos. Si el operando es 0 y el registro es `%r0`, la instrucción se comporta como una instrucción nop, lo que significa que no se realiza operación alguna.

Ejemplo de uso: `sethi 0x304F15, %r1`

Significado: Cargar en los 22 bits más significativos de `%r1` el valor hexadecimal 304F15, colocando en cero los 10 bits menos significativos.

Código objeto: `0000 0011 0011 0000 0100 1111 0001 0101`

Instrucción: `andcc`

Descripción: Producto lógico, bit por bit, de los operandos origen, almacenando el resultado en el operando de destino. Los códigos de condición se fijan en función del resultado.

Ejemplo de uso: `andcc %r1, %r2, %r3`

Significado: Realizar el producto lógico de los registros `%r1` y `%r2` y almacenar el resultado en `%r3`.

Código objeto: `1000 0110 1000 1000 0100 0000 0000 0010`

Instrucción: `orcc`

Descripción: Suma lógica, bit por bit, de los operandos origen, almacenando el resultado en el operando de destino. Los códigos de condición se fijan en función del resultado.

Ejemplo de uso: `orcc %r1, 1, %r1`

Significado: Colocar en 1 el bit menos significativo del registro `%r1`.

Código objeto: `1000 0010 1001 0000 0110 0000 0000 0001`

Instrucción: `orncc`

Descripción: Negación de la suma lógica, bit por bit, de los operandos origen, almacenando el resultado en el operando de destino. Los códigos de condición se fijan en función del resultado.

Ejemplo de uso: `orncc %r1, %r0, %r1`

Significado: Complementar el registro `%r1`.

Código objeto: `1000 0010 1011 0000 0100 0000 0000 0000`

Instrucción: `srl`
Descripción: Desplaza a la derecha el contenido de un registro en una cantidad de cero a 31 posiciones. Las posiciones que quedan vacías sobre la izquierda del registro desplazado se completan con ceros.
Ejemplo de uso: `srl %r1, 3, %r2`
Significado: Desplazar el contenido del registro `%r1` tres posiciones a la derecha y almacenar el resultado en `%r2`. Los tres bits más significativos de `%r2` se completan con ceros.
Código objeto: `1000 0101 0011 0000 0110 0000 0000 0011`

Instrucción: `addcc`
Descripción: Suma los operandos origen dejando el resultado en el operando destino, para lo cual utiliza representación signada de complemento a dos. Los códigos de condición se establecen en función del resultado.
Ejemplo de uso: `addcc %r1, 5, %r1`
Significado: Sumar 5 al contenido del registro `%r1`.
Código objeto: `1000 0010 1000 0000 0110 0000 0000 0101`

Instrucción: `call`
Descripción: Invoca una subrutina y almacena la dirección de la instrucción actual (la que contiene el llamado) en `%r15`, lo que implica una operación de "llamado y enlace". En el código ensamblado, el campo `disp30` del formato CALL contiene un desplazamiento de 30 bits a partir de la dirección de la instrucción `call`. La dirección de la siguiente instrucción a ser ejecutada se obtiene sumando 4 · `disp30` (lo que desplaza `disp30` a los 30 bits más significativos de la dirección de 32 bits) con la dirección de la instrucción en ejecución. Nótese que `disp30` puede ser negativo.
Ejemplo de uso: `call sub_r`
Significado: Invocar una subrutina que comienza en la posición de memoria `sub_r`. Para el código objeto indicado a continuación, `sub_r` se encuentra ubicada en memoria 25 palabras (100 bytes) después de la instrucción `call`.
Código objeto: `0100 0000 0000 0000 0000 0000 0001 1001`

Instrucción: `jmpl`
Descripción: Salta y vincula (retorno de subrutina). Salta a una nueva dirección y almacena la dirección de la instrucción actual (la que corresponde a la instrucción `jmpl`) en el registro de destino.
Ejemplo de uso: `jmpl %r15 + 4, % r0`
Significado: Retorno de subrutina. El valor del contador de programa al momento de la instrucción `call` quedó almacenado en `%r15` y, por consiguiente, la dirección de retorno debe calcularse como la que sigue al `call`, en `%r15 + 4`. La dirección de la instrucción actual se descarta en `%r0`.
Código objeto: `1000 0001 1100 0011 1110 0000 0000 0100`

Instrucción: be
Descripción: Si el código de condición z es 1, salta a la dirección que se obtiene sumándole 4 · disp22 del formato de la instrucción de salto a la dirección de la instrucción actual. Si el bit z del registro de estado es 0, el control se transfiere a la instrucción siguiente al be.
Ejemplo de uso: `be label`
Significado: Saltar a `label` si el código de condición z vale 1. Para el ejemplo, se supone que `label` se encuentra en una dirección de memoria cinco palabras (20 bytes) mayor que la de la instrucción de be.
Código objeto: `0000 0010 1000 0000 0000 0000 0000 0101`

Instrucción: bneg
Descripción: Si el código de condición n es 1, salta a la dirección que se obtiene sumándole 4 · disp22 a la dirección de la instrucción actual. Si el bit n del registro de estado es 0, el control se transfiere a la instrucción siguiente al bneg.
Ejemplo de uso: `bneg label`
Significado: Saltar a `label` si el código de condición n vale 1. Para el ejemplo, se supone que `label` se encuentra en una dirección de memoria cinco palabras (20 bytes) mayor que la de la instrucción de bneg.
Código objeto: `0000 1100 1000 0000 0000 0000 0000 0101`

Instrucción: bcs
Descripción: Si el código de condición c es 1, salta a la dirección que se obtiene sumándole el valor 4 · disp22 a la dirección de la instrucción actual. Si el bit c del registro de estado es 0, el control se transfiere a la instrucción siguiente al bcs.
Ejemplo de uso: `bcs label`
Significado: Saltar a `label` si el código de condición c vale 1. Para el ejemplo, se supone que `label` se encuentra en una dirección de memoria cinco palabras (20 bytes) mayor que la de la instrucción de bcs.
Código objeto: `0000 1010 1000 0000 0000 0000 0000 0101`

Instrucción: bvs
Descripción: Si el código de condición v es 1, salta a la dirección que se obtiene sumándole el valor 4 · disp22 a la dirección de la instrucción actual. Si el bit v del registro de estado es 0, el control se transfiere a la instrucción siguiente al bvs.
Ejemplo de uso: `bvs label`
Significado: Saltar a `label` si el código de condición v vale 1. Para el ejemplo, se supone que `label` se encuentra en una dirección de memoria cinco palabras (20 bytes) mayor que la de la instrucción de bvs.
Código objeto: `0000 1110 1000 0000 0000 0000 0000 0101`

Instrucción: ba

Descripción: Saltar a la dirección obtenida sumando a la dirección de la instrucción actual el valor 4 · `disp22` incluido en la instrucción de salto.

Ejemplo de uso: `ba label`

Significado: Saltar a `label` independientemente de los valores que adopten los códigos de condición. Para el ejemplo, se supone que `label` se encuentra en una dirección de memoria cinco palabras (20 bytes) menor que la de la instrucción de ba.

Código objeto: `0001 0000 1011 1111 1111 1111 1111 1011`

4.3 Directivas (seudo operaciones)

Además de las instrucciones descriptas, que pueden ser soportadas por la arquitectura ARC, también existen directivas (**seudo operaciones**) que no son códigos de operación, sino más bien instrucciones dirigidas al ensamblador para que realice ciertas acciones en el momento del ensamble. La figura 4.12 muestra una lista de directivas y ejemplos. Nótese que a diferencia de los códigos de operación de las instrucciones, que son específicos para una máquina determinada, el tipo y la naturaleza de las directivas son específicas de un determinado ensamblador, dado que es el ensamblador quien las ejecuta.

La directiva `.equ` (igualar) instruye al ensamblador para que iguale un valor o una cadena de caracteres con un símbolo, de modo que el símbolo pueda ser utilizado a lo largo el programa como si en su lugar estuviesen escritos el valor o la cadena de caracteres. Las directivas `.begin` (comienzo) y `.end` (final) le indican al ensamblador dónde debe comenzar y terminar, respectivamente, el proceso de traducción. Cualquier sentencia que aparezca antes del `.begin` o después del `.end` será ignorada. La representación simbólica de cualquier programa puede incluir más de un par `.begin/.end`, pero tiene que existir un `.end` por cada `.begin`, y debe haber al menos un `.begin`. El uso de las directivas `.begin` y `.end` es útil para el proceso de depuración, porque permite que ciertas partes del programa se hagan invisibles al ensamblado.

La directiva `.org` (origen) hace que la instrucción siguiente se ensamble suponiendo que se la ubicará, en el momento de la ejecución, en la posición de memoria especificada (2048 en el caso de la figura 4.12). La directiva `.dwb` (definir bloque de palabras) reserva un bloque de palabras de cuatro bytes, generalmente para arreglos. El **contador de posiciones** (que lleva la cuenta de cuál es la instrucción que está siendo ensamblada por el programa traductor) se desplaza hasta ubicarlo delante del bloque, de acuerdo con la cantidad de posiciones obtenidas al multiplicar por cuatro el argumento de la directiva `.dwb`, ya que dicho argumento especifica la cantidad de palabras necesarias.

Las directivas `.global` y `.extern` tienen que ver con nombres de variables y direcciones que se definen en un módulo de código a ensamblar y se utilizan en otro. La directiva `.global` hace accesible un rótulo dado para que pueda ser usado en otros módulos. La directiva `.extern` identifica un rótulo que está siendo usado en el módulo

local y ha sido definido en otro módulo (rótulo que en el otro módulo en cuestión debería estar indicado como `.global`). En el próximo capítulo se analizará el uso de estas dos directivas. Las directivas `.macro`, `.endmacro`, `.if` y `.endif` también se analizan en el capítulo siguiente.

Directiva	Forma de uso	Función o significado
`.equ`	`X .equ #10`	Asignar a X el valor $(10)_{16}$
`.begin`	`.begin`	Comienzo de traducción
`.end`	`.end`	Fin de traducción
`.org`	`.org 2048`	Cambiar el contador de posición a 2048
`.dwb`	`.dwb 25`	Reservar un bloque de 25 palabras
`.global`	`.global Y`	La variable Y se usa en otro módulo
`.extern`	`.extern Z`	La variable Z se define en otro módulo
`.macro`	`.macro M a, b, ...`	Definir macroinstrucción M. Parámetros formales: a, b, ...
`.endmacro`	`.endmacro`	Fin de definición de macroinstrucción
`.if`	`.if <cond>`	Ensamblar solo si <cond> es cierta
`.endif`	`.endif`	Fin de estructura condicional

Figura 4.12 • Directivas para el lenguaje ensamblador de ARC.

4.4 Ejemplos de programación en lenguaje ensamblador

El proceso de escritura de un programa en lenguaje simbólico es similar al proceso de escritura de dicho programa en lenguaje de alto nivel, excepto por el hecho de que muchos detalles que en el lenguaje de alto nivel se vuelven abstractos, son totalmente explícitos en el lenguaje simbólico. En esta sección se analizan dos ejemplos de programas escritos en el lenguaje simbólico de ARC.

Programa: sumar dos enteros

Considérese la escritura de un programa para el lenguaje de programación simbólico de ARC que sume los enteros 15 y 9. Una posible codificación es la que se muestra en la figura 4.13. El programa comienza y termina con un par de directivas `.begin`/`.end`. La directiva `.org` le indica al programa ensamblador que la traducción debe iniciarse de modo tal que el código ensamblado se cargue en memoria a partir de la dirección 2048. Los operandos 15 y 9 se almacenan en las variables `x` e `y`, respectivamente. Solo se pueden sumar números que estén almacenados en registros de ARC (dado que solamente `ld` y `st` pueden acceder a memoria) y, por consiguiente, el programa se inicia cargando en los registros `%r1` y `%r2` las variables `x` e `y`. La instrucción `addcc` suma `%r1` con `%r2`

y deja el resultado en el registro `%r3`. La instrucción `st` se encarga de almacenar `%r3` en la dirección de memoria `z`. La instrucción `jmpl` con operandos `%r15 + 4, %r0` provoca un retorno a la instrucción siguiente del programa que invocó a la rutina de suma. Debe notarse que en caso de que no haya habido programa usuario que invocara al programa de suma, se entenderá como que este último ha sido invocado por el sistema operativo. Las variables `x`, `y` y `z` se definen a continuación del programa.

```
! Este programa suma dos números
        .begin
        .org 2048
progl:  ld      [x], %r1          ! Cargar el registro %r1 con el valor de x
        ld      [y], %r2          ! Cargar el registro %r2 con el valor de y
        addcc   %r1, %r2, %r3     ! %r3 ← %r1 + %r2
        st      %r3, [z]          ! Guardar el contenido el registro %r3 en Z
        jmpl    %r15 + 4, %r0     ! Retorno
x:      15
y:      9
z:      0
        .end
```

Figura 4.13 • Un programa para ARC que suma dos enteros.

En la práctica, el código SPARC equivalente al código ARC de la figura 4.13 no es del todo correcto. Las instrucciones `ld`, `st` y `jmpl` demoran al menos dos ciclos de instrucción para completarse, y dado que SPARC inicia una nueva instrucción en cada pulso de reloj, estas instrucciones deben estar seguidas por instrucciones que no dependan de sus resultados. Esta propiedad, que permite iniciar una nueva instrucción antes de haber completado la anterior se conoce como **segmentación** (*pipelining*), y se analiza con detalle en el capítulo 10.

Programa: suma de un arreglo de enteros

Considérese ahora un programa algo más complejo, que suma un arreglo de enteros. La figura 4.14 ilustra una posible codificación. Como en el caso del ejemplo anterior, el programa se inicia y se termina con un par `.begin/.end`. La directiva `.org` indica que la traducción se debe realizar de modo que el código ensamblado se cargue en memoria a partir de la dirección 2048. Se utiliza una directiva para definir el símbolo `a_start`, al que se le asigna el valor 3000.

El programa se inicia cargando la longitud del arreglo `a`, la que se presenta en bytes, en el registro `%r1`. El programa carga luego la dirección de comienzo del arreglo `a` en `%r2` y pone en cero el registro `%r3`, que acumulará el resultado parcial. El registro `%r3` se pone en cero a través de una operación de producto lógico con el registro `%r0`, que siempre contiene el valor cero. Cualquier producto lógico que se realice entre el registro `%r0` y cualquier otro registro dará siempre por resultado el valor cero.

```
! Este programa suma tantos números como lo indica LENGTH
! Utilización de registros:   %r1 - Longitud del arreglo a
!                             %r2 - Dirección de comienzo del arreglo a
!                             %r3 - Suma parcial
!                             %r4 - Puntero al arreglo a
!                             %r5 - Contiene un elemento de a
          .begin              ! Comienzo de traducción
          .org  2048          ! Comienzo del programa en 2048
a_start   .equ  3000          ! Dirección del arreglo a
          ld    [length], %r1 ! %r1 ← longitud del arreglo a
          ld    [address],%r2 ! %r2 ← dirección del arreglo a
          andcc %r3, %r0, %r3 ! %r3 ← 0
loop:     andcc %r1, %r1, %r0 ! Verificar número de elementos restantes
          be    done          ! Finaliza cuando LENGTH = 0
          addcc %r1, -4, %r1  ! Decrementar tamaño del arreglo
          addcc %r1, %r2, %r4 ! Dirección del próximo elemento
          ld    %r4, %r5      ! %r5 ← dirección de memoria [%r4]
          addcc %r3, %r5, %r3,! Sumar elemento nuevo al contenido de %r3
          ba    loop          ! Repetir el lazo

done:     jmpl  %r15 + 4, %r0 ! Retorno a rutina principal
length:         20            ! El arreglo a tiene 5 números (20 bytes)
address:        a_start
          .org  a_start       ! Comienzo del arreglo a
a:               25           ! A continuación los elementos del arreglo
                -10
                 33
                 -5
                  7
          .end                ! Fin de traducción
```

Figura 4.14 • Un programa para ARC que suma cinco enteros.

El rótulo `loop` inicia un lazo que suma los sucesivos elementos del arreglo a dejando la suma parcial en el registro `%r3`, en cada iteración. El lazo se inicia verificando si el número restante de elementos a sumar (`%r1`) es cero. Para obtener este dato realiza el producto lógico de `%r1` consigo mismo, con lo que logra acomodar los códigos de condición. El código que interesa es `z`, el cual se pondrá en 1 si `%r1 = 0`. Las banderas restantes (`n`, `v` y `c`) se acomodan de acuerdo con el resultado. El valor de `z` se analiza por medio de la instrucción `be`. Si no quedan más elementos a sumar dentro del arreglo, el programa bifurca a `done`, de donde vuelve a la rutina que lo invocó (y que puede ser el sistema operativo, si este es el nivel más alto del programa usuario).

Si luego de la verificación de `%r1 = 0` el programa no hubiese salido del lazo, se decrementa el valor de `%r1` en cuatro (el tamaño de cada palabra medido en bytes) para lo cual se le suma al contenido de `%r1` el valor `-4`. La dirección de comienzo del arreglo a (almacenada en `%r2`) y el índice en `a` (`%r1`) se suman en `%r4`, el que apunta a un nuevo elemento de `a`. El elemento al que apunta `%r4` se carga en `%r5`, y su valor se suma a la suma parcial (`%r3`). La instrucción "`ba loop`" lleva al programa nuevamente al comienzo

del lazo cerrado. La variable `length` se almacena luego de las instrucciones. Los cinco elementos del arreglo `a` se ubican en el área de memoria especificado por el argumento de la directiva `.org` (dirección 3000).

Nótese que hay tres instrucciones para calcular la dirección del elemento siguiente en el arreglo, dada la dirección del elemento inicial en `%r2` y la longitud del arreglo en bytes en `%r1`:

```
addcc %r1, -4, %r1  ! apuntar al nuevo elemento a sumar
addcc %r1, %r2, %r4 ! sumarlo a la base del arreglo
ld %r4, %r5         ! cargar en %r5 el próximo elemento
```

Este método para el cálculo de la dirección de una variable como la suma de una base más un índice se usa tan frecuentemente que tanto ARC como otros lenguajes simbólicos tienen modos especiales de direccionamiento para utilizarlo. En el caso de ARC, la dirección de la instrucción `ld` se calcula como la suma de dos registros o la de un registro con una constante de 13 bits. Nótese que el registro `%r0` siempre contiene el valor cero, por lo que al especificar `%r0`, lo que se hace en forma implícita en la instrucción `ld`, se desperdicia la oportunidad de permitir que la misma instrucción `ld` haga el cálculo de la dirección. Un registro único puede contener la dirección del operando y permite resolver en dos instrucciones lo que en el ejemplo anterior lleva tres:

```
addcc %r1, -4, %r1  ! apuntar al próximo elemento a ser sumado
ld %r1 + %r2, %r5 ! cargar el próximo elemento en %r5
```

Nótese que también se evita el uso de un registro, `%r4`, antes utilizado como elemento de almacenamiento temporario de la dirección.

4.4.1 Variantes en las arquitecturas y en los direccionamientos

La arquitectura de ARC es la típica arquitectura de una máquina de carga y descarga. Los programas escritos para este tipo de máquinas se ejecutan generalmente en menos tiempo, en parte debido a la reducción del tráfico entre CPU y memoria, porque los operandos se cargan solo una vez en la CPU y los resultados se salvan en memoria únicamente cuando se completa el cálculo. El aumento en el tamaño de memoria requerida por el programa se considera un precio que vale la pena pagar.

No era tal el caso cuando las memorias eran varios órdenes de magnitud más caras que hoy y los procesadores eran varios órdenes de magnitud más limitados que hoy, lo que representa el cuadro de las primeras épocas de la computación. En aquellas condiciones, cuando la CPU probablemente ofrecía solo un único registro para almacenar valores aritméticos, los resultados intermedios debían almacenarse en memoria. Estas máquinas presentaban instrucciones aritméticas de **una**, **dos** y **tres** direcciones. Esto sig-

nifica que una instrucción podía realizar cálculo aritmético con tres, dos o uno de sus resultados u operandos en memoria, lo que se contrapone con la arquitectura ARC, en la que todas las operaciones aritméticas y lógicas requieren que sus operandos estén almacenados en registros.

Considérese la forma en que podría evaluarse la sentencia A = B x C + D a través de los distintos tipos de instrucciones de tres, dos y una dirección. En los ejemplos que siguen, la referencia a la variable A significa, en realidad, "el operando cuya dirección es A". Para poder realizar algunas estadísticas de rendimiento de los diversos fragmentos del programa, pueden hacerse las siguientes suposiciones:

- Las palabras de dirección y datos son de 16 bits, o 2 bytes, tamaño habitual en máquinas de aquellas épocas.
- Los códigos de operación tienen 8 bits.

Se determinarán tanto el tamaño como el tráfico de memoria del programa, en bytes, con estas suposiciones.

El tráfico de memoria tiene dos componentes, por un lado la instrucción en sí misma, que debe traerse de memoria a la CPU para poder ser ejecutada, y por otro los datos: operandos que deben trasladarse a la CPU para realizar la operación y resultados que deben ser devueltos a memoria cuando se completa el cálculo. La observación de estos cálculos permitirá visualizar algunos de los acuerdos entre el tamaño del programa y el tráfico de memoria que ofrecen los distintos tipos de instrucciones. El lector deberá tener en cuenta que los códigos de operación, como `mult` y `add`, son genéricos: no son instrucciones de ARC.

Instrucciones de tres direcciones

En una instrucción de tres direcciones, A = B x C + D podría codificarse como:

```
mult B, C, A          ! A ← B x C
add D, A, A           ! A ← A + D
```

lo que implica multiplicar B por C y almacenar el resultado en A. (En este punto del programa, A se usa como elemento de almacenamiento para resultados temporarios.) Luego se sumará el valor de A con D, y se volverá a almacenar el resultado en la dirección A. Cada instrucción tiene un tamaño de 1 + 2 + 2 + 2 = 7 bytes. Por consiguiente, el tamaño del programa es de 14 bytes. El tráfico de memoria para cada instrucción se calcula como la suma de la cantidad de búsquedas de instrucciones, 7 bytes, más el tráfico de datos, 2 + 2 + 2 = 6 bytes, lo que lleva a un total de 13 bytes por instrucción. En consecuencia, el tráfico total de memoria del programa es de 26 bytes.

Instrucciones de dos direcciones

En una instrucción de dos direcciones, uno de los operandos queda sobrescrito por el resultado. Por consiguiente, el código para la expresión A = B x C + D puede ser:

```
load    B, A        ! A ← B
mult    C, A        ! A ← A x C
add     D, A        ! A ← A + D
```

Cada una de las tres instrucciones ocupa ahora 1 + 2 + 2 = 5 bytes, por lo que el tamaño del programa es de 15 bytes. En el cálculo del tráfico de memoria debe notarse que cada una de las búsquedas de las instrucciones implicará un tráfico de 5 bytes, tal como se ha dicho. No obstante, el tráfico de datos de la primera instrucción será de 2 palabras, o sea, 4 bytes, en tanto que el tráfico de datos de las dos instrucciones restantes será de 3 palabras, o sea, 6 bytes. Esto se debe a que cada una de las dos últimas instrucciones requiere cargar ambos operandos en la CPU, lo que requiere 2 palabras, o 4 bytes, en tanto que el almacenamiento del resultado en memoria requiere una palabra de dos bytes. En consecuencia, el tráfico total de las tres instrucciones será de (5 + 4) + (5 + 6) + (5 + 6) = 31 bytes.

Instrucciones de una dirección

Las instrucciones con una sola dirección emplean un registro aritmético único de la CPU, accesible al programador, conocido como **acumulador**. El acumulador, normalmente, contiene un operando aritmético y sirve como lugar de almacenamiento del resultado de las operaciones aritméticas. El formato de una dirección no es de uso común en estos días, pero lo era en épocas anteriores de la computación, en las que los registros eran más caros y frecuentemente cumplían con funciones diversas. El acumulador sirve para el almacenamiento temporario tanto de uno de los operandos como del resultado. El código para la expresión A = B x C + D es ahora

```
load    B           ! Acc ← B
mult    C           ! Acc ← Acc x C
add     D           ! Acc ← Acc + D
store   A           ! A ← Acc
```

La instrucción `load` carga B en el acumulador, `mult` realiza la multiplicación de C por el contenido del acumulador y deja el resultado en el mismo acumulador, y `add` realiza la suma requerida. La instrucción `store` almacena el acumulador en A. Cada instrucción ocupa 3 bytes, por lo que el tamaño total del programa es de 3 x 4 = 12 bytes. Nótese que cada instrucción requiere la búsqueda o almacenamiento de solo una palabra de dos bytes. Por lo tanto, el tráfico de memoria para cada instrucción es de 3 + 2 = 5 bytes y el tráfico total de memoria del programa es de 4 x 5 = 20 bytes.

Nótese que la máquina de una dirección ofrece el menor tamaño de programa y el menor tráfico de instrucciones de todos los ejemplos vistos. Esto explica la popularidad de las viejas computadoras basadas en acumulador, en las épocas en que los registros y las memorias costaban cientos de dólares por bit.

Registros de propósitos especiales

Además de los registros de uso general y de los acumuladores descriptos previamente, la mayoría de las arquitecturas modernas incluyen otros registros que se dedican a propósitos específicos. Como ejemplo de este tipo de registros pueden mencionarse:

- Registros índice para memoria, como el caso de los registros índice de origen (SI, *source index*) e índice de destino (DI, *destination index*) en el caso del 80x86. Estos registros se utilizan como punteros al comienzo o al final de un arreglo almacenado en memoria. Algunas instrucciones especiales referidas a "cadenas" transfieren un byte o una palabra desde una dirección de comienzo de memoria, a la que apunta SI, hacia una dirección final de memoria, a la que apunta DI, y posteriormente incrementan o decrementan los registros para que apunten al byte siguiente o a la palabra siguiente.
- Registros de punto flotante. Muchos procesadores de la generación actual ofrecen registros e instrucciones especiales para el manejo de números de punto flotante.
- Registros para manejo de tiempo y de operaciones de temporización. El procesador PowerPC 601 tiene registros asociados con un reloj de tiempo real. El mismo permite la medición de tiempo real con alta resolución, con el objeto de indicar fecha y hora del día. El rango que ofrecen es de, aproximadamente, 135 años, con una resolución de 128 ns.
- Registros de apoyo al sistema operativo. Muchos procesadores modernos ofrecen registros para manejo del sistema de memoria.
- Registros accesibles solo a través de "instrucciones privilegiadas" o en el "modo supervisor". Con el objeto de evitar daños accidentales o intencionales al sistema, muchos procesadores incluyen instrucciones y registros especiales que son inaccesibles para los usuarios y las aplicaciones convencionales. Estas instrucciones y registros solo pueden ser usados por el sistema operativo.

4.4.2 Eficiencia de las arquitecturas de programación

Si bien las estadísticas sobre tamaño de programa y uso de memoria calculadas en los ejemplos anteriores están planteadas fuera del contexto de los programas en los que podrían haber estado contenidos los ejemplos, muestran claramente que aun el hecho de tener un solo registro como elemento temporario de almacenamiento en la CPU puede tener un efecto muy significativo sobre la eficiencia en la ejecución del programa. De hecho, el procesador Pentium Intel, considerado entre los más veloces dentro de los procesado-

res para aplicaciones generales, tiene un solo acumulador, aun cuando tiene una cantidad de registros de uso especial como elementos de apoyo. Existen muchos otros factores que afectan la eficiencia de un conjunto de instrucciones, tales como el tiempo que necesita la instrucción para ejecutar su función, o la velocidad a la cual puede funcionar el procesador.

4.5 El acceso a la información en la memoria. Modos de direccionamiento

Hasta este momento, se han analizado cuatro métodos para obtener la dirección de un dato almacenado en memoria:

- Un valor constante, conocido al momento del ensamble.
- El contenido de un registro.
- La suma de los contenidos de dos registros.
- La suma del contenido de un registro más una constante.

Modo de direccionamiento	Sintaxis	Significado
Inmediato	#K	K
Directo	K	M[K]
Indirecto	(K)	M[M[K]]
Indirecto a través de registro	(Rn)	M[Rn]
Indexado a través de registro	(Rm + Rn)	M[Rm + Rn]
Basado en registro	(Rm + X)	M[Rm + X]
Indexado basado en registro	(RM + Rn + X)	M[Rm + Rn + X]

Tabla 4.1 • Modos de direccionamiento.

La tabla 4.1 le asigna nombres a estos modos de direccionamiento y muestra algunas otras alternativas. Nótese que la sintaxis de la tabla difiere de la de ARC. Esta es una característica particular, común y desafortunada de los lenguajes simbólicos. Cada uno de ellos difiere de los demás en sus convenciones sintácticas. La notación M[x] en la columna de significados supone que la memoria es un arreglo, M, indexado según una dirección que se obtiene a través del cálculo indicado entre corchetes. Parece haber un surtido injustificado de modos de direccionamiento, pero cada uno de ellos tiene su aplicación:

- El direccionamiento inmediato permite hacer referencia a una constante cuyo valor se conoce al momento del ensamble.

- El direccionamiento directo se utiliza para acceder a aquellos datos cuya dirección se conoce al momento del ensamble.
- El direccionamiento indirecto se utiliza para acceder a una variable puntero cuya dirección se conoce al momento de la compilación. Los procesadores modernos casi no usan este modo de direccionamiento debido a que requiere dos referencias a memoria para acceder al operando, lo que complica la instrucción que la utiliza. El programador que desee usar este direccionamiento para acceder a datos necesita utilizar dos instrucciones, una para acceder al puntero y otra para llegar al valor al que dicho puntero hace referencia. Esto tiene como beneficio el hecho de exponer la complejidad del modo de direccionamiento, con lo que tal vez se logra desalentar el uso del mismo.
- El direccionamiento indirecto a través de registro se usa cuando la dirección del operando no se conoce hasta el momento de la ejecución. El almacenamiento de operandos en una pila cumple con esta descripción, por lo que se accede a dichos operandos a través de un direccionamiento indirecto vía registro, habitualmente en la forma de instrucciones del tipo "*push*" y "*pop*" que, además, decrementan e incrementan, respectivamente, el registro puntero.
- El direccionamiento indexado, el basado en registro y el indexado basado en registro se usan para acceder a los componentes de arreglos tales como el que se muestra en la figura 4.14. y a aquellos componentes ubicados más allá del comienzo de la pila, en una estructura conocida como bloque de datos (***stack frame***), la que se analiza en la sección siguiente.

4.6 Acceso a subrutinas y pilas

Una **subrutina**, conocida a veces como **función** o **procedimiento**, es una secuencia de instrucciones a la que se invoca como si se tratara, desde la visión de alto nivel, de una única instrucción. Cuando un programa llama a una subrutina, se transfiere el control del programa a la subrutina, la que ejecuta la secuencia de instrucciones requerida, tras lo cual vuelve a la posición inmediatamente siguiente a la que generó el llamado. Existen distintos métodos para pasar argumentos hacia y desde la rutina invocada, a los que suele denominarse **convenciones de llamada**. El proceso de transferencia de argumentos entre subrutinas suele conocerse como **enlace entre subrutinas**.

Una de las convenciones de llamada de subrutinas simplemente coloca los argumentos en registros. El código de la figura 4.15 muestra un programa que carga dos argumentos en los registros `%r1` y `%r2`, invoca la subrutina `add_1` y recupera el resultado desde el registro `%r3`. La subrutina `add_1` toma sus operandos desde `%r1` y `%r2` y deja el resultado en `%r3` antes de retornar por medio de la instrucción `jmpl`. El método es sencillo y veloz, pero no funciona si la cantidad de argumentos a ser transferidos excede el número de registros disponibles, o si las llamadas a subrutinas están fuertemente encadenadas.

```
! Rutina invocante
        .
        .
        .
    ld      [x], %r1
    ld      [y], %r2
    call    add_1
    st      %r3, [z]
        .
        .
        .
x:  53
y:  10
z:   0
```

```
! Rutina invocada
! %r3 ← %r1 + %r2


add_1:  addcc   %r1, %r2, %r3
        jmpl    %r15 + 4, %r0
```

Figura 4.15 • Llamado a subrutina utilizando registros.

Una segunda forma de llamada crea lo que se denomina una **zona de transferencia de datos**. La dirección de la zona de transferencia de datos se entrega a la rutina invocada en un registro predeterminado. La figura 4.16 ilustra un ejemplo de este método de llamada. La directiva .dwb que aparece en la rutina invocante genera una zona de transferencia de datos, de tres palabras, en las direcciones x, x + 4 y x + 8. La rutina invocante carga sus dos argumentos en x y x + 4, llama a la subrutina add_2, y luego recupera el resultado que le es transferido desde add_2 en la dirección de memoria x + 8. La dirección x correspondiente al comienzo de la zona de datos se le transfiere a la subrutina add_2 en el registro %r5.

```
! Rutina invocante
        .
        .
        .
    st      %r1, [x]
    st      %r2, [x+4]
    sethi   x, %r5
    srl     %r5, 10, %r5
    call    add_2
    ld      [x+8], %r3
        .
        .
        .
! Zona de transferencia de datos
x:  .dwb    3
```

```
! Rutina invocada
! x[2] ← x[0] + x[1]

add_2:  ld      %r5, %r8
        ld      %r5 + 4, %r9
        addcc   %r8, %r9, %r10
        st      %r10, %r5 + 8
        jmpl    %r15 + 4, %r0
```

Figura 4.16 • Llamado a subrutina usando una zona para transferencia de datos.

Nótese que la instrucción sethi requiere una constante para definir su operando origen, por lo que el ensamblador reconoce la estructura de sethi incluida en el programa invocante de la subrutina y reemplaza x por su dirección. La instrucción srl que sigue a sethi coloca la dirección x en los 22 bits menos significativos de %r5, dado que sethi coloca su operando en los 22 bits más significativos del registro utilizado. Un método alternativo para cargar la dirección de x en %r5 consistiría en el uso de una dirección de

memoria para almacenar la dirección de `x`, y el posterior uso de la instrucción `ld` para cargar la dirección en `%r5`. Si bien esta última alternativa es más sencilla, la utilización de `sethi/srl` es más rápida debido a que no involucra operaciones de acceso a memoria que consumen tiempo.

La subrutina `add_2` lee sus dos operandos desde el área de datos en las direcciones `%r5` y `%r5 + 4`, y coloca el resultado en la misma zona de datos en la dirección `%r5 + 8` antes de retornar. El uso de una zona para transferencia de datos permite el pasaje de bloques de datos de todo tamaño sin tener que copiar más que un registro en el proceso de llamado a la subrutina. Sin embargo, cuando se trabaja con rutinas recursivas, este sistema puede generar algunos problemas complicados, debido a que una rutina que se invoque a sí misma requerirá varias zonas de transferencia de datos. Las zonas de transferencia de datos tienen la ventaja de que su tamaño puede ser ilimitado, pero tienen también la desventaja de que dicho tamaño debe ser conocido al momento del ensamble.

Una tercera convención para las llamadas a subrutinas utiliza una pila. La idea general es que la rutina invocante coloca (empuja) todos sus argumentos (o punteros a los mismos, si los datos tienen tamaños grandes) en una pila secuencial del tipo LIFO (*last in first out*). La rutina invocada extrae de la pila los argumentos transferidos, a la vez que coloca en la misma cualquier valor que deba retornar. El programa invocante, a su vez, rescata de la pila los valores devueltos por la rutina invocada y continúa con la ejecución. El registro de la CPU conocido como **puntero a la pila** (*stack pointer*) contiene la dirección de la cabeza de la pila. Muchas máquinas tienen instrucciones del tipo `push` y `pop` que, automáticamente, decrementan e incrementan el puntero a la pila cuando se colocan elementos en la pila o cuando se los extrae de la misma.

```
  ! Rutina invocante
      .
      .
      .
%sp  .equ   %r14
     addcc  %sp, -4, %sp
     st     %r1, %sp
     addcc  %sp, -4, %sp
     st     %r2, %sp
     call   add_3
     ld     %sp, %r3
     addcc  %sp, 4, %sp
      .
      .
      .
```

```
! Rutina invocada
! Los argumentos están en la pila
! %sp[0] ← %sp[0] + %sp[4]
    %sp  .equ   %r14
add_3:  ld      %sp, %r8
        addcc   %sp, 4, %sp
        ld      %sp, %r9
        addcc   %r8, %r9, %r10
        st      %r10, %sp
        jmpl    %r15 + 4, %r0
```

Figura 4.17 • Llamado a subrutina utilizando una pila.

La ventaja del uso de una pila es que el tamaño de la misma aumenta o disminuye a medida que resulta necesario. Esto admite la posibilidad de encadenar llamadas a procedimientos, en cantidad arbitraria, sin tener que declarar el tamaño de la pila en el momento del ensamble. La figura 4.17 ilustra un ejemplo de transferencia de argumentos por

medio de una pila. El registro `%r14` sirve como puntero a la pila (`%sp`), el que se inicializa desde el sistema operativo con anterioridad a la ejecución de la rutina que lo requiere. La rutina invocante coloca sus argumentos `%r1` y `%r2` en la pila a través del decremento del puntero a la pila (lo que ubica al registro `%sp` apuntando a la primera palabra libre por encima de la pila) tras lo cual se almacena cada argumento en la nueva posición apuntada en la pila. Se invoca a la subrutina `add_3`, la que rescata sus argumentos de la pila, realiza la operación de suma y deja su resultado en la pila antes de retornar. La rutina invocante recupera su argumento de la primera dirección de la pila y continúa su ejecución.

Cualquiera sea la convención de llamada, se utiliza para el llamado la instrucción `call`, la que rescata el valor actual del contador de programa en `%r15`. Cuando la subrutina completa su ejecución, necesita volver a la dirección de la instrucción siguiente a la que efectuó la llamada, la que se encuentra una palabra (cuatro bytes) después del valor rescatado del PC. Por consiguiente, la sentencia "`jmpl %r15 + 4, %r0`" completa el retorno. No obstante, si la rutina invocada llama a su vez a otra rutina, el valor del contador de programa que había sido guardado originalmente en `%r15` se verá sobrescrito por la llamada encadenada, lo que implica que no se podrá retornar correctamente al programa original a través de `%r15`. Con el objeto de permitir llamadas y retornos encadenados, el valor actual de `%r15` (llamado **registro de enlace**) debería rescatarse en la pila, junto con cualquier otro registro que requiera ser recuperado luego del retorno.

Si se utiliza una convención de llamada basada en registros, antes de proceder a una llamada encadenada debería rescatarse el registro de enlace en alguno de los registros disponibles. Si se utiliza una zona de transferencia de datos, la misma debería disponer de espacio reservado para el registro de enlace. Si se usa una pila, el registro de enlace también debe apilarse. Para cada una de las convenciones utilizadas para llamadas a subrutinas, antes de encadenar subrutinas deben rescatarse los contenidos del registro de enlace y las variables locales. En caso contrario, cuando se produzca una llamada encadenada a la misma subrutina se producirá la pérdida de las variables locales.

Existen muchas variantes de las técnicas de llamadas básicas, pero, probablemente, la más popular para la utilización de subrutinas sea la técnica basada en el uso de una pila. Cuando se utiliza un método de llamado a subrutina que requiere el uso de una pila para invocar subrutinas anidadas, se construye un **área de pila** para que contenga los argumentos que se han de transferir a una rutina invocada, la dirección de retorno a la rutina invocante y cualquier variable local. La figura 4.18 ilustra un programa escrito en lenguaje de alto nivel que requiere el uso de rutinas anidadas. La función del programa no es importante, como tampoco lo es el hecho de haber usado C como lenguaje de programación. Lo que sí importa es la forma en que han sido implementados los llamados a subrutinas.

La figura 4.19 ilustra la conducta de la pila a lo largo del programa. El programa principal invoca a `func_1` con los argumentos 1 y 2, y luego invoca a `func_2` con el argumento 10 antes de completar su ejecución. La función `func_1` posee dos variables loca-

```
Línea
No.  /* Programa en C que ilustra el llamdo de subrutinas
     encadenadas (anidadas)*/
00   main ()
01   {
02        int w,z;              /* Variables locales */
03        w = func_1 (1,2); /* Llamada subrutina func_1 */
04        z = func_2 (10);  /* Llamada subrutina func_2 */
05   }                          /* Fin de programa principal */

06   func_1 (x,y)               /* Calcular x * x + y */
07   int x,y;              /* Parâmetros a tranferir a func_1 */
08   {
09        int i,j;              /* Variables locales */
10        i = x * x;
11        j = i + y;
12        return (j);   /* Devolver j al programa principal */
13   }

14   func_2 (a)                 /* Calcular a * a + a + 5 */
15   int a;                /* Parâmetro a transferir a func_2 */
16   {
17        int m,n;              /* Variables locales */
18        n = a + 5;
19        m = func_1 (a,n);
20        return (m);   /* Devolver m al programa principal */
21   }
```

Figura 4.18 • Programa escrito en C que ilustra llamados encadenados.

les llamadas `i` y `j`, que se usan para el cálculo del valor `j` a devolver. La función `func_2` posee dos variables locales `m` y `n` que se utilizan para crear el argumento `m` que la función devuelve antes de retornar.

El puntero de pila (`%r14` por convención, pero se lo denominará `%sp`) se inicializa antes del comienzo de la ejecución del programa, función que suele estar a cargo del sistema operativo. El compilador es el elemento responsable de implementar la técnica de llamada y, por consiguiente, será el compilador quien genere el código que envíe los parámetros y la dirección de retorno a la pila, reservando espacio en la pila para las variables locales e invirtiendo el proceso a medida que las rutinas regresan a quienes las llamaron. La conducta de la pila, que se ilustra en la figura 4.19, se produce entonces como consecuencia de la ejecución de código generado por el compilador, pero dicho código bien pudo haber sido escrito directamente en lenguaje de máquina.

Cuando se inicia la ejecución del programa, el puntero apunta al elemento superior de la pila del sistema (figura 4.19a). Cuando el programa principal invoca a `func_1` en la línea 03 del programa de la figura 4.18, utilizando los argumentos 1 y 2, estos argumentos se salvan en la pila, como se muestra en la figura 4.19b. El control se transfiere a la subrutina `func_1` a través de una instrucción de `call` (la que no se muestra). La rutina `func_1` rescata de la pila la dirección de retorno, la que se encuentra en `%r15` como resultado de la instrucción de `call` (figura 4.19c). Se reserva espacio en la pila (figura 4.19d) para las

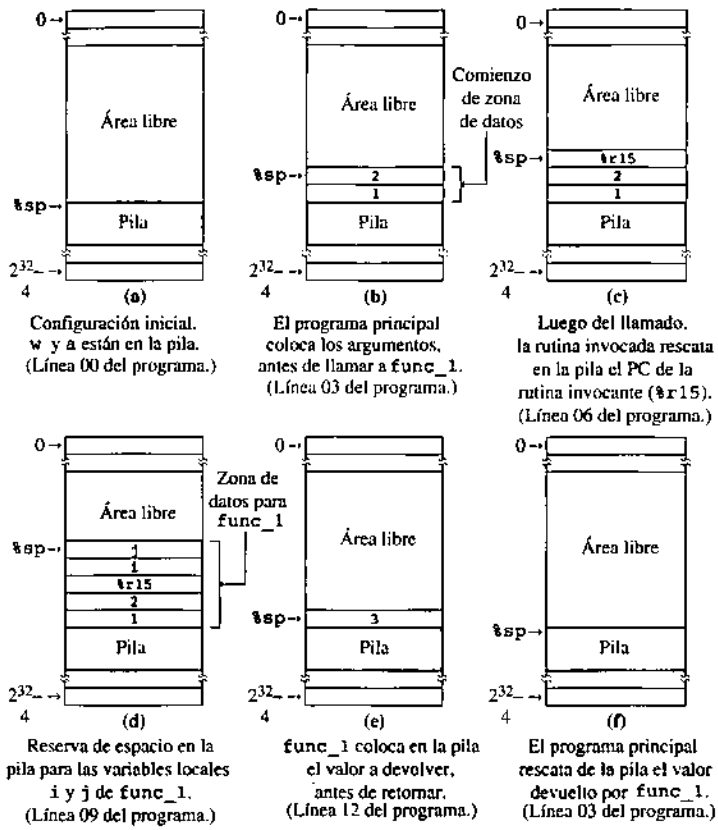
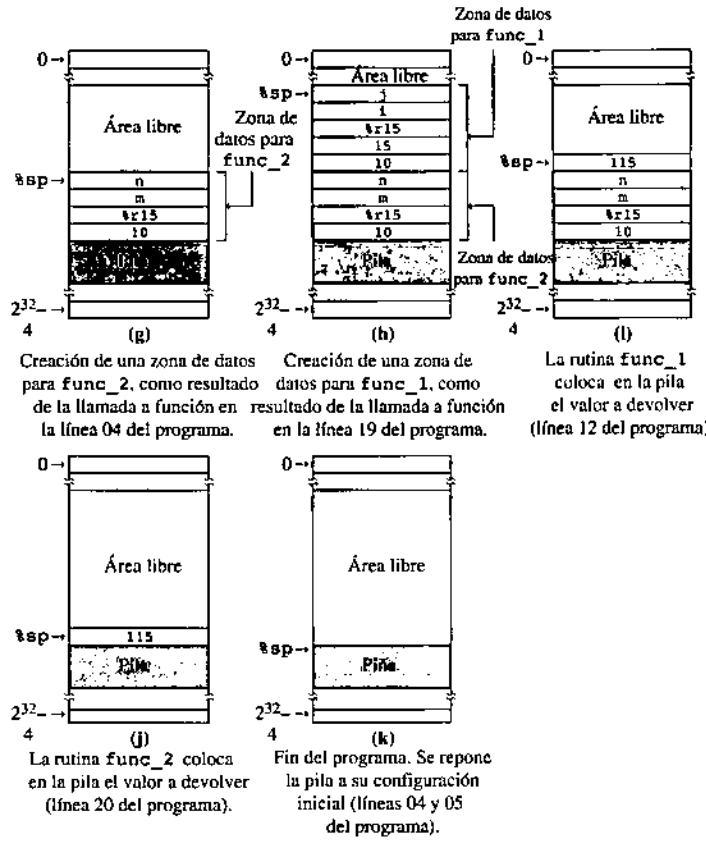


Figura 4.19 • (a-f) Conducta de la pila durante la ejecución del programa de la figura 4.18.

variables locales `i` y `j` de `func_1`. A esta altura, la llamada a la rutina `func_1` dispone, en la pila, de un bloque compuesto por los argumentos que se transfieren a `func_1`, la dirección de retorno al programa principal y las variables locales de `func_1`.

Inmediatamente antes de que `func_1` vuelva a la rutina que la llamó, libera el espacio ocupado en la pila por sus variables locales, recupera de la pila la dirección de retorno y coloca en la pila el valor que debe regresar, tal como se ve en la figura 4.19e. La instrucción `jmpl` devuelve el control a la rutina invocante, la que es responsable de recuperar de la pila el valor transferido desde `func_1`, y de decrementar el puntero de pila a su posición original anterior a la llamada, tal como se ve en la figura 4.19f. Luego, se ejecuta la rutina `func_2`, con lo que se reinicia el proceso de generación del bloque o zona de datos en la pila, tal como se muestra en la figura 4.19g. Dado que la rutina `func_2` realiza un llamado a `func_1` antes de retornar al programa principal, existirán en la pila, en forma simultánea, sendos bloques para cada una de las rutinas `func_1` y `func_2`, tal como se ve en la figura 4.19h. El proceso se completa en la misma forma anterior, reubicándose finalmente el puntero de pila en su posición original, de acuerdo con lo que se muestra en las figuras 4.19i-k.



Creación de una zona de datos para `func_2`, como resultado de la llamada a función en la línea 04 del programa.

Creación de una zona de datos para `func_1`, como resultado de la llamada a función en la línea 19 del programa.

La rutina `func_1` coloca en la pila el valor a devolver (línea 12 del programa).

La rutina `func_2` coloca en la pila el valor a devolver (línea 20 del programa).

Fin del programa. Se repone la pila a su configuración inicial (líneas 04 y 05 del programa).

Figura 4.19 • (g-k)

4.7 Entrada y salida en lenguaje simbólico

Por último, se llega al análisis de las formas que permiten que un programa escrito en lenguaje de máquina pueda comunicarse con el mundo exterior: las actividades de entrada y salida (E/S). Una de las formas en que puede administrarse la comunicación entre los dispositivos de entrada-salida y el resto de la máquina requiere el uso de instrucciones especiales y de un bus de entrada-salida especialmente dedicado a este objetivo. El método alternativo para la interacción con los dispositivos de entrada-salida implica el concepto de entrada-salida mapeada en memoria principal, en el cual los dispositivos de entrada-salida ocupan espacios del mapa de direcciones de memoria en los que no existe memoria físicamente disponible. El acceso a los dispositivos se realiza como si fueran posiciones de memoria y, por consiguiente, no hay necesidad de crear nuevas instrucciones para el manejo de los dispositivos.

Como ejemplo del concepto de entrada-salida mapeada en memoria principal puede considerarse nuevamente el mapa de memoria de ARC, que se representa en la figura 4.20. Se observan algunas nuevas zonas de memoria, reservadas para dos módulos de

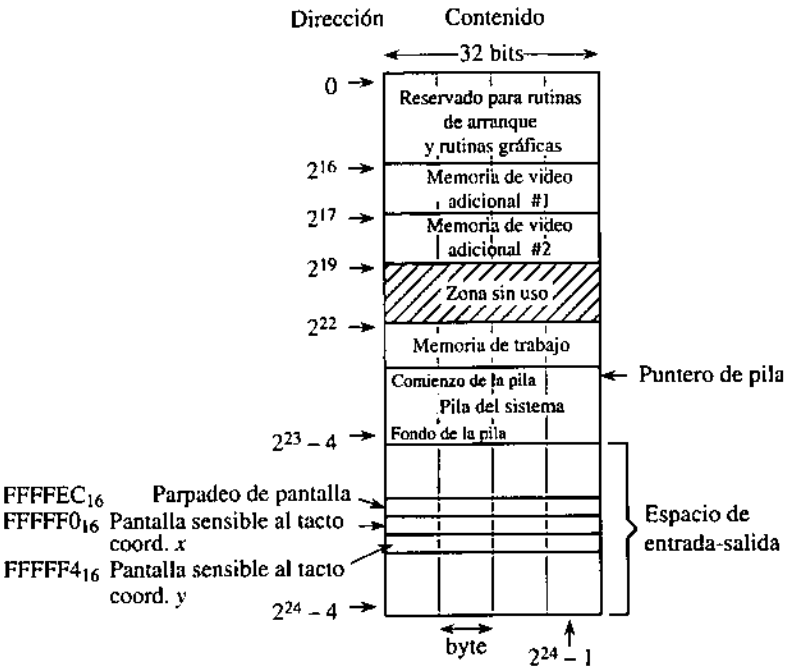


Figura 4.20 • Mapa de memoria de ARC que incluye la distribución de los elementos en memoria.

memoria de video adicionales y para una **pantalla sensible al tacto**. Existen dos tipos de pantallas sensibles al tacto, la fotónica y la eléctrica. La figura 4.21 ilustra una versión de la pantalla fotónica. La pantalla se encuentra cubierta, tanto en horizontal como en vertical, por una serie de rayos o haces. Si algo (un dedo, por ejemplo) interrumpe un haz, puede determinarse la posición a través de los haces interrumpidos. En una versión alternativa de la pantalla, el elemento de visualización se encuentra cubierto por una superficie sensible al tacto. El usuario debe hacer contacto con la pantalla para poder efectuar una selección.

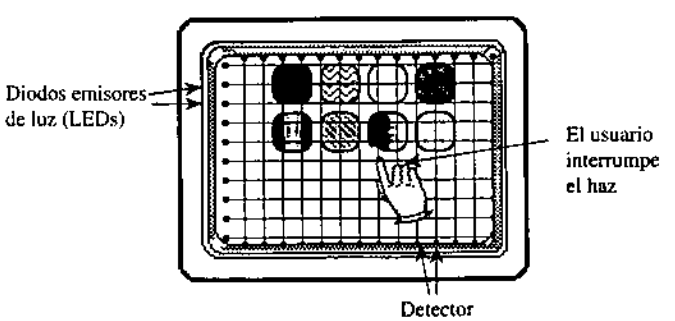


Figura 4.21 • Selección de un objeto en una pantalla sensible al tacto.

La única memoria físicamente existente en el sistema ocupa el espacio de direcciones entre 2^{22} y $2^{23} - 1$. (Debe recordarse que $2^{23} - 4$ es la dirección del byte más significativo de la palabra más alta en el formato de almacenamiento *big-endian*). El resto del espacio de direcciones está ocupado por otros componentes. El área de direcciones que va desde 0 hasta $2^{16} - 1$, inclusive, contiene programas residentes para la operación de arranque de encendido (*power-on bootstrap*) y para rutinas gráficas básicas. El espacio de direcciones entre 2^{16} y $2^{19} - 1$ se utiliza para ubicar dos módulos adicionales de memoria de video, los que serán analizados en el problema 4.3. Debe notarse que solo se tendrá información válida disponible en esas direcciones cuando los módulos de memoria estén físicamente ubicados en el sistema.

Por último, el espacio de direcciones entre 2^{23} y $2^{24} - 1$ se utiliza para los dispositivos de entrada-salida. En este sistema, las coordenadas X e Y que determinan la posición seleccionada por el usuario se actualizan automáticamente en registros ubicados dentro del mapa de memoria. Los contenidos de estos registros se obtienen simplemente por medio de la lectura de las direcciones de memoria en las que se encuentran. La posición "parpadeo de pantalla" hará que la pantalla parpadee cada vez que se la escriba.

Si se desea escribir un programa simple que haga parpadear la pantalla cuando el usuario cambia de posición en la misma, el diagrama de flujo de la figura 4.22 ilustra una forma de resolverlo. Se leen primero los registros X e Y, tras lo cual se los compara con sus valores anteriores. Si se modifica alguna de las posiciones, se produce el parpadeo de la pantalla, se actualizan los valores previos de X e Y y se repite el proceso. Si no se produjo cambio alguno en las coordenadas, el proceso simplemente se repite. Este ejemplo representa el método de acceso a un dispositivo de entrada-salida por programa. (Véase el problema 4.3, al final del capítulo, para una descripción más detallada.)

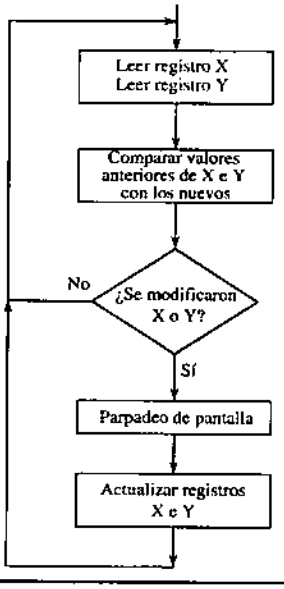


Figura 4.22 • Diagrama de flujo que ilustra la estructura de control de un programa que verifica una pantalla sensible al tacto.

4.8 Estudio de un caso: la arquitectura de programación de la máquina virtual Java

Java es un lenguaje de programación de alto nivel desarrollado por Sun Microsystems, que ha tomado una posición prominente entre los programadores. Un aspecto clave es que los códigos binarios de Java son independientes de la plataforma, lo que significa que el mismo código compilado puede ejecutarse sin modificación alguna sobre cualquier computadora que soporte la **máquina virtual Java** (JVM, *Java Virtual Machine*). La máquina virtual Java es el medio por el que Java logra ser independiente de las plataformas: se implementa una especificación normalizada de JVM en el conjunto primario de instrucciones de las máquinas subyacentes, y así los códigos Java compilados funcionan en cualquier ambiente JVM.

Aquellos programas escritos en lenguajes totalmente compilados, como C, C++ o Fortran, se compilan hacia el código nativo de la arquitectura de destino y, por ende, en general no pueden transportarse entre plataformas a menos que el código fuente sea compilado nuevamente para la nueva plataforma. Los lenguajes interpretados, como Perl, Tcl, AppleScript y shell script son en gran medida independientes de la plataforma, pero pueden llegar a ejecutarse en tiempos 100 a 200 veces mayores que los lenguajes compilados. Los programas Java se compilan en un formato intermedio conocido como ***bytecodes***, los que se ejecutan en tiempos diez veces mayores que los lenguajes completamente compilados. No obstante, la compatibilidad a través de las diversas plataformas y otras características del lenguaje hacen de Java un lenguaje de programación ventajoso para muchas aplicaciones.

La figura 4.23 ilustra una visión de alto nivel de la arquitectura JVM. JVM es una máquina basada en pilas, lo que significa que los operandos se colocan y se extraen de una pila, en lugar de ser transferidos entre registros de uso general. No obstante, hay una cantidad de registros dedicados, y también una cantidad de variables locales que cumplen con la función de los registros de uso general en la arquitectura "real" (no virtual). El motor de ejecución de Java toma los códigos compilados de Java en su entrada y los interpreta en una implementación software de JVM, o los ejecuta directamente en una implementación circuital de la misma.

La figura 4.24 muestra una implementación Java del programa SPARC analizado en la figura 4.13. La figura ilustra tanto el programa fuente de Java como los *bytecodes* en los que fue compilado. El archivo de *bytecodes* se conoce como un **archivo de clase** de Java (denominación que reciben los programas Java compilados).

Si se analiza un archivo de clase, se podrá ver que solo algunos pocos bytes del mismo contienen instrucciones. El resto del archivo es un encabezado que debe estar incluido en el archivo para permitir su ejecución sobre JVM. En la figura 4.25 se ha procedido a "desensamblar" el programa código nuevamente hacia su formato de alto nivel. Las posiciones del código están expresadas en hexadecimal, a partir de la dirección `0x00`. Los primeros cuatro bytes contienen el número mágico `0xcafebabe`, el que identifica al

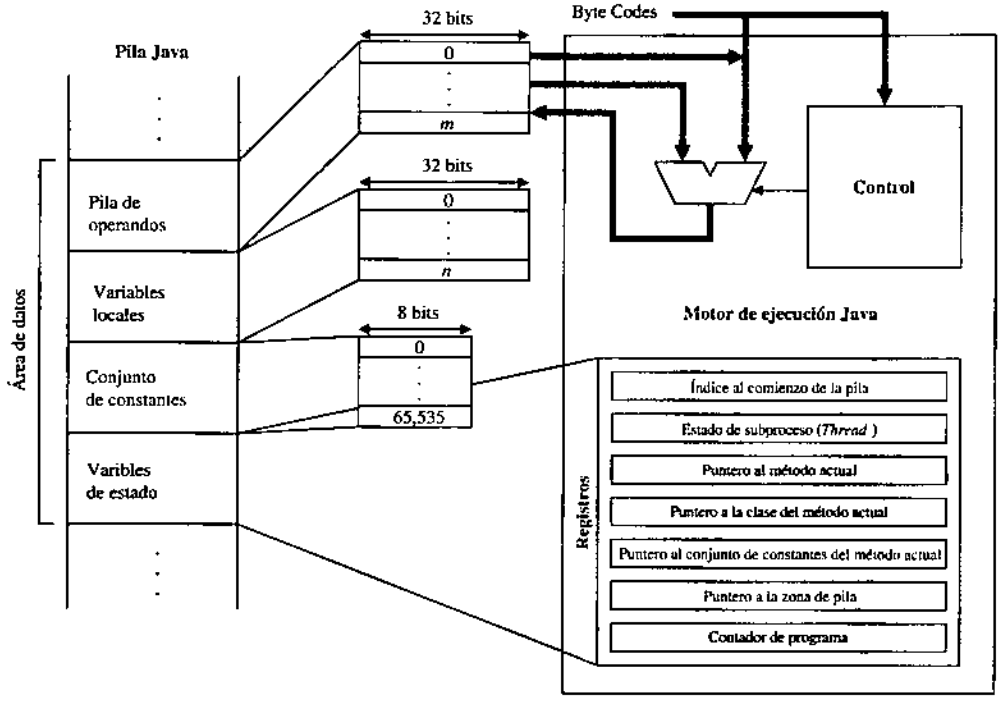


Figura 4.23 • Arquitectura de la máquina virtual Java.

programa como un archivo de clase de Java compilado. Los números correspondientes a mayor y menor versión se refieren a la versión de Java para el cual se ha compilado el programa. Viene a continuación la cantidad de entradas al **conjunto de constantes** (*constant pool*), que para este ejemplo es 17. La primera entrada (dirección cero del conjunto de constantes) se reserva siempre para JVM y no se incluye en el archivo de clase, si bien la indexación en el conjunto de constantes se inicia siempre en la posición 0, como si estuviese explícitamente representada. El conjunto de constantes contiene los nombres de **métodos** (funciones), atributos y toda otra información que sea requerida por el sistema que administra el momento de la ejecución.

El resto del archivo está compuesto mayormente por el conjunto de constantes y por instrucciones ejecutables de Java. No se analizan aquí todos los detalles referidos a los archivos de clase. Para el lector que desee una descripción completa del formato de archivos de clase de Java se recomienda la lectura de J. Meyer y T. Downing.

El código real que corresponde al programa fuente Java, que simplemente suma las constantes 15 y 9 y devuelve el resultado (24) a la rutina invocante en la pila, aparece en las direcciones `0x00e3–0x00ef`. Debe tenerse en cuenta, a medida que se sigue el ejemplo, que las variables `x`, `y`, `z` de la figura 4.24 se definen como enteros, los que en Java se definen con un tamaño de 32 bits. La figura 4.26 muestra cómo se interpreta esa porción de los *bytecodes*. El programa convierte las constantes 15 y 9 en palabras de 32 bits mediante la extensión del signo, y coloca las palabras en la pila usando como

```
// Este es el archivo add.java

public class add {
    public static void main(String args[]) {
        int x=15, y=9, z=0;
        z = x + y;
        }
    }
```

```
0000 cafe babe 0003 002d 0012 0700 0e07 0010   ..............
0010 0a00 0200 040c 0007 0005 0100 0328 2956   ............()V
0020 0100 1628 5b4c 6a61 7661 2f6c 616e 672f   ...([Ljava/lang/
0030 5374 7269 6e67 3b29 5601 0006 3c69 6e69   String;)V...<ini
0040 743e 0100 0443 6f64 6501 000d 436f 6e73   t>...Code...Cons
0050 7461 6e74 5661 6c75 6501 000a 4578 6365   tantValue...Exce
0060 7074 696f 6e73 0100 0f4c 696e 654e 756d   ptions...LineNum
0070 6265 7254 6162 6c65 0100 0e4c 6f63 616c   berTable...Local
0080 5661 7269 6162 6c65 7301 000a 536f 7572   Variables...Sour
0090 6365 4669 6c65 0100 0361 6464 0100 0861   ceFile...add...a
00a0 6464 2e6a 6176 6101 0010 6a61 7661 2f6c   dd.java...java/l
00b0 616e 672f 4f62 6a65 6374 0100 046d 6169   ang/Object...mai
00c0 6e00 2100 0100 0200 0000 0000 0200 0900   n.!.............
00d0 1100 0600 0100 0800 0000 2d00 0200 0400   ..........-.....
00e0 0000 0d10 0f3c 1009 3d03 3e1b 1c60 3eb1   .....<..=.>..`>.
00f0 0000 0001 000b 0000 000e 0003 0000 0004   ................
0100 0008 0006 000c 0002 0001 0007 0005 0001   ................
0110 0008 0000 001d 0001 0001 0000 0005 2ab7   ..............*.
0120 0003 b100 0000 0100 0b00 0000 0600 0100   ................
0130 0000 0100 0100 0d00 0000 0200 0f00        ..............
```

Figura 4.24 • Programa Java y archivo compilado.

intermediarios a las variables locales 0 y 1. En un alarde de gimnasia innecesaria, el programa extrae los dos valores desde la pila hacia dos variables locales, e inmediatamente los vuelve a empujar hacia la pila. El programa invoca luego la instrucción `iadd`, la que extrae los dos elementos superiores de la pila, los suma y coloca el resultado en la cabeza de la pila, tras lo cual procede a retornar.

Las operaciones innecesarias que se realizan sobre la pila representan algunas de las razones por las que JVM corre 10 veces más despacio que el código nativo. El programa empuja los argumentos hacia la pila, los saca luego hacia dos variables locales 1 y 2, y los vuelve a introducir en la pila antes de sumarlos. Para los compiladores de otros lenguajes, esta transferencia es vista como redundante por lo que, en dichos casos, sería eliminada. Dado este único ejemplo, probablemente exista un amplio espacio para mejorar la velocidad de ejecución con respecto a la velocidad de ejecución de los programas JVM de hoy en día, que es unas 10 veces menor que la de los otros lenguajes. Otras mejoras pueden presentarse en la forma de lo que se denomina compiladores ***just in time*** (JIT), literalmente compiladores puntuales o justo a tiempo. Estos compiladores, en lugar de interpretar los *bytecodes* JVM uno por uno cada vez que aparecen, aprovechan el hecho de que la mayoría de los programas dedican la mayor parte de su tiempo a realizar lazos y otras rutinas iterativas. A medida que el compilador JIT encuentra cada línea de código

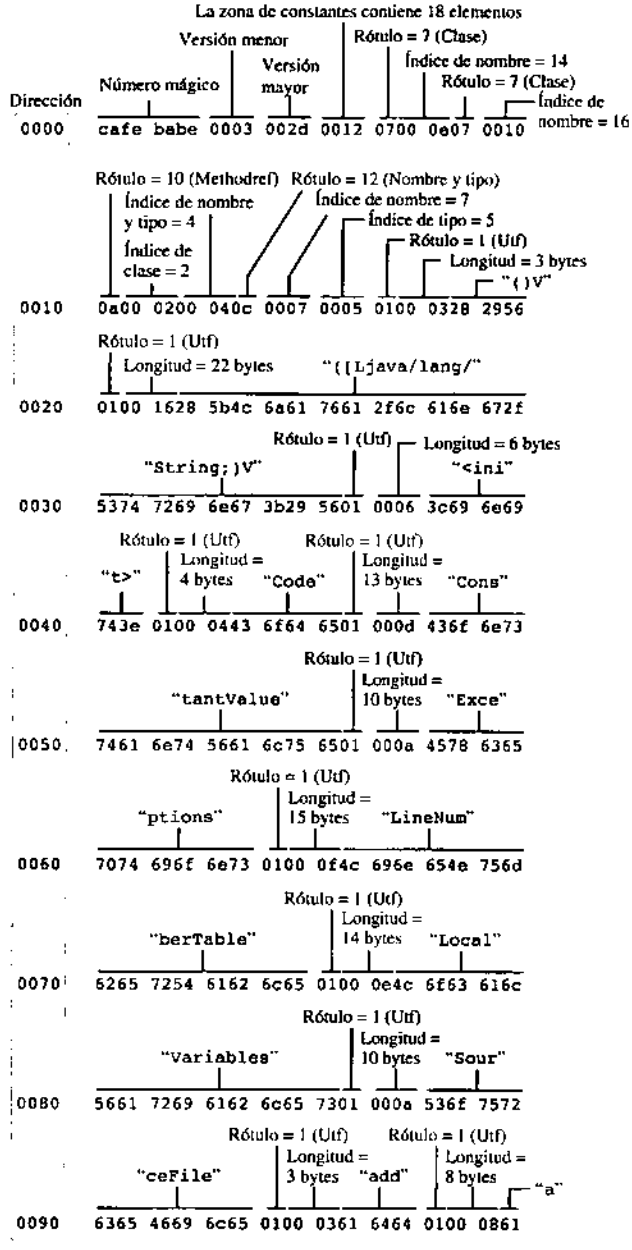


Figura 4.25 • Un archivo de clase de Java.

por primera vez, lo traduce al lenguaje nativo y lo almacena en memoria para un eventual uso posterior. La próxima vez que el código debe ser ejecutado, se ejecuta la forma nativa, compilada, en lugar de los *bytecodes*.

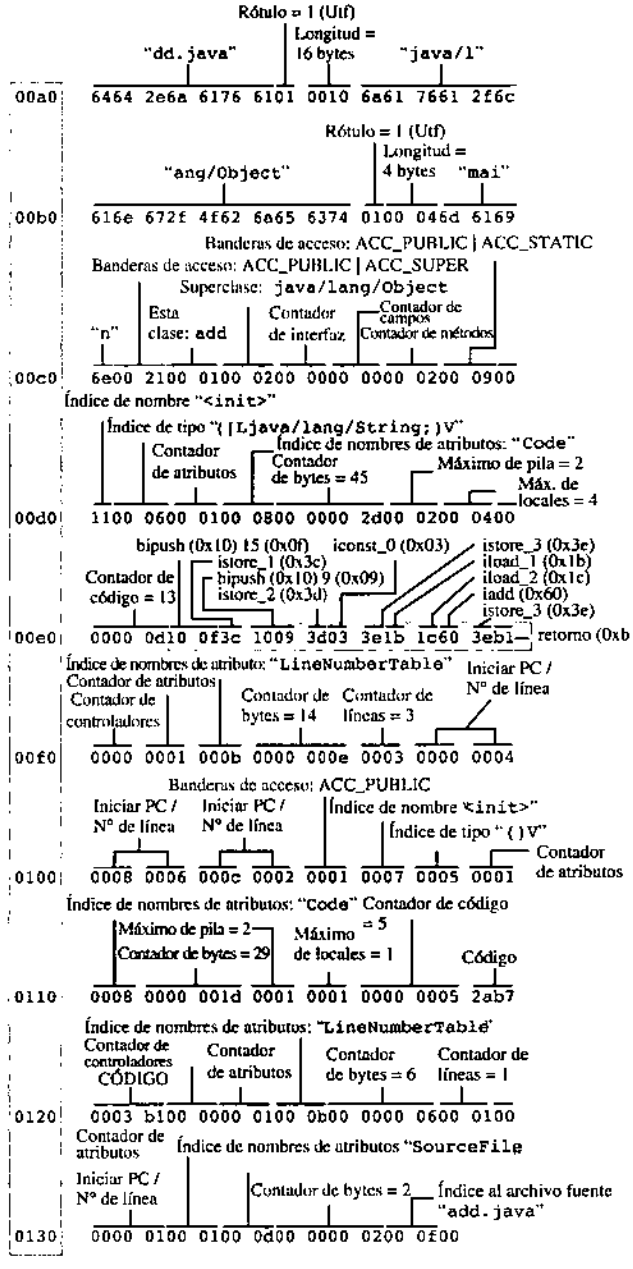


Figura 4.25 • Continuación.

Posición	Código	Mnemónico	Significado
0x00e3	0x10	bipush	Colocar próxima palabra en la pila
0x00e4	0x0f	15	Argumento para bipush
0x00e5	0x3c	istore_1	Extraer de la pila la variable local 1
0x00e6	0x10	bipush	Colocar próxima palabra en la pila
0x00e7	0x09	9	Argumento para bipush
0x00e8	0x3d	istore_2	Extraer de la pila la variable local 2
0x00e9	0x03	iconst_0	Colocar en la pila la constante 0
0x00ea	0x3e	istore_3	Extraer de la pila la variable local 3
0x00eb	0x1b	iload_1	Colocar en la pila la variable local 1
0x00ec	0x1c	iload_2	Colocar en la pila la variable local 2
0x00ed	0x60	iadd	Sumar los dos elementos superiores de la pila
0x00ee	0x3e	istore_3	Extraer de la pila la variable local 3
0x00ef	0xb1	return	Retornar

Figura 4.26 • Versión desensamblada del código que implementa el programa Java de la figura 4.24.

Resumen

En este capítulo se ha presentado la arquitectura de programación ARC y se han estudiado algunas propiedades generales de las arquitecturas de programación. En el diseño de un juego de instrucciones se debe lograr un balance entre el rendimiento del sistema y las características de la tecnología en la que se implementa el procesador. La interacción entre la CPU y la memoria es una consideración clave.

La forma en que se calcula la dirección cuando se realiza un acceso a memoria se denomina modo de direccionamiento. Se han examinado las secuencias de cálculo que pueden combinarse para implementar un modo de direccionamiento. También se han analizado algunos casos específicos comúnmente identificados por nombre.

Se han analizado, además, las distintas partes de un sistema de computación que juegan algún papel en la ejecución de un programa. Se ha podido determinar que un programa está construido con secuencias de instrucciones, las que se obtienen del juego de instrucciones de la CPU. En el capítulo siguiente se analizará la forma en que estas secuencias de instrucciones se traducen en código objeto.

Para lectura posterior

El material de este capítulo es, en su mayor parte, un resumen de la experiencia acumulada a lo largo de cincuenta años de diseño de computadoras de programa almacenado. Si bien cada generación de sistemas de computación se identifica con una tecnología circuital específica, han existido también algunas arquitecturas de programación históricamente importantes. En los sistemas de primera generación de los años 1950, como la EDVAC de von Neuman, la UNIVAC de Eckert y Mauchly, y la IBM 701, la programación se realizaba a mano en lenguaje de máquina. Aun siendo simples, es-

tas arquitecturas de programación definieron los conceptos fundamentales respecto de códigos de operación y operandos.

El concepto de una arquitectura de programación como entidad con identificación propia puede encontrarse en los diseñadores del sistema IBM S/360. Las arquitecturas VAX de Digital Equipment Corporation pueden rastrear sus propias raíces en este periodo, en el que se llevaba a cabo el desarrollo de las minicomputadoras PDP-4 y PDP-8. Tanto la 360 como la VAX son arquitecturas de dos direcciones. Las arquitecturas más significativas de una sola dirección incluyen al procesador Intel 8080, antecesor de la moderna arquitectura 80x86, y a su contemporáneo Z80 de Zilog. Como arquitectura de cero direcciones es de significación histórica la computadora Burroughs B5000.

Hay gran cantidad de referencias que cubren los distintos lenguajes de máquina existentes. Son demasiados para ser detallados aquí y, en consecuencia, solo se mencionan algunos pocos de los casos más celebres. Los lenguajes de máquina de las máquinas de Babbage se detallan en el texto de A. G. Bromley. El lenguaje de máquina de la vieja computadora del Institute for Advanced Study (IAS) está tratado por W. Stallings. G. W. Struble analiza el lenguaje de máquina de la IBM 360, en tanto que el del procesador 68000 se puede encontrar en A. Gill y otros, y el de SPARC en el texto correspondiente. Una descripción completa de la máquina virtual Java y el formato de archivos de clase de Java fue desarrollado por J. Meyer y T. Downing.

Bromley, A. G., "The Evolution of Babbage's Calculating Engines", en: *Annals of the History of Computing*, vol. 9, 1987, p.p. 113-138.

Gill, A., E. Corwin y A. Logar, *Assembly Language Programming for the 68000*, Prentice Hall, 1987.

Meyer, J. y T. Downing, *Java Virtual Machine*, O'Reilly & Associates, 1997.

SPARC International, Inc., *The SPARC Architecture Manual: Version 8*, Prentice Hall, 1992.

Stallings, W., *Computer Organization and Architecture*, 4ª ed., Prentice Hall, 1996. (Traducción al español disponible: *Organización y arquitecura de computadores*, 5ª ed., Prentice Hall, 2000.)

Struble, G. W., *Assembler Language Programming: The IBM System/360 and 370*, 2ª ed., Addison Wesley, 1975.

Problemas

4.1 Una memoria tiene 2^{24} posiciones direccionables. ¿Cuál es la menor cantidad de bits requerida en la palabra de direcciones que permite direccionar todas las 2^{24} posiciones?

4.2 ¿Cuáles son la primera y la última dirección de una memoria de 2^{20} bytes si la menor estructura direccionable es una palabra de cuatro bytes?

4.3 En la figura 4.20 se muestra el mapa de memoria de ARC.

a. ¿Qué cantidad de memoria (en bytes) tiene disponible para cada uno de los módulos de memoria de video adicionales? (Se pide la respuesta en la forma de potencias de 2 o suma de potencias de 2, por ejemplo, 2^{10}.)

b. Si se desplaza un dedo por sobre la pantalla sensible al tacto, las posiciones horizontal (x) y vertical (y) del movimiento se actualizan en registros que se encuentran en las direcciones $(\text{FFFFF0})_{16}$ y $(\text{FFFFF4})_{16}$, respectivamente. Cuando se escribe un 1 en el registro accesible en la dirección $(\text{FFFFEC})_{16}$, la pantalla parpadea, tras lo cual la estructura de hardware pone en cero la dirección $(\text{FFFFEC})_{16}$ sin que el programa deba realizar dicha operación. Escribir un programa ARC que haga parpadear la pantalla cada vez que cambie la posición del usuario. Utilizar como esqueleto del programa el que se muestra a continuación:

```
        begin
        ld [x], %r7 ! los registros %r7 y %r8 apuntan a las coordenadas
        ld [y], %r8 ! x e y de la pantalla
        ld [flash], %r9 !el registro %r9 apunta a la posición de parpadeo
loop: ld %r7, %r1 ! Cargar los valores actuales x e y de la
        ld %r8, %r2 ! posición de pantalla en %r1 y %r2
        ld [old_x], %r3 !Cargar los valores anteriores x e y de la posición
        ld [old_y], %r4 ! de pantalla en %r3 y %r4
        orncc %r3, %r0, %r3 ! Complementar a 1 el valor old_x
        addcc %r3, 1, %r3 ! Complementar a 2 el valor old_x
        addcc %r1, %r3, %r3 ! %r3 <- x - old_x
        be x_not_moved ! Saltar si no hubo cambio n x
        ba moved ! x cambió, no hace falta controlar y
x_not_moved: ! Ingresar código de programa aquí,
        ! alrededor de cuatro líneas

                        <- CÓDIGO DE USUARIO AQUÍ

   ! Se accede a esta parte del código
   ! solamente si se movió el cursor de la pantalla
   ! Parpadeo de pantalla, almacenar nuevos valores de x e y, repetir
   moved: orcc %r0, 1, %r5 ! Cargar el registro %r5 con 1
   st      %r5, %r9 ! Guardar el 1 en el registro de parpadeo
   st      %r1 [old_x] ! Actualizar los valores anteriores
   st      %r2 [old_y] ! de la posición
   ba      loop ! Repetir
   flash: #FFFFC ! Posición del registro de parpadeo
   x: #FFFFC0 ! Posición del registro de coordenada x
   y: #FFFFEC4 ! Posición del registro de coordenada y
   old_x: 0 ! Valor anterior de x
   old_y: 0 ! Valor anterior de y

.end
```

4.4 Escribir una subrutina para ARC que realice una operación de intercambio entre los operandos de 32 bits `x = 25` e `y = 50`, los que se encuentran almacenados en memoria. Usar la menor cantidad posible de registros.

4.5 A continuación se muestra una sección de código simbólico de ARC. ¿Qué función cumple el programa? ¿Suma números o limpia algo? ¿Simula un lazo **for**, un lazo **while** o alguna otra cosa? Suponer que `a` y `b` son posiciones de memoria que se encuentran definidas en algún otro lugar del código.

```
Y:   ld    [k], %r1
     addcc %r1, -4, %r1
     st    %r1, [k]
     bneg  X
     ld    [a], %r1, %r2
     ld    [b], %r1, %r3
     addcc %r2, %r3, %r4
     st    %r4, %r1, [c]
     ba    Y
X:    jmpl %r15 + 4, %r0
k:    40
```

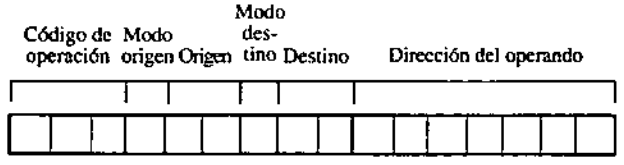
4.6 Un sistema buscador contiene un pequeño procesador con una memoria de 2^7 palabras de ocho bits. La arquitectura de programación tiene cuatro registros: R0, R1, R2 y R3. El conjunto de instrucciones se muestra en la figura 4.27, así como los códigos binarios correspondientes a cada registro, al formato de instrucción y a los **modos**, los que determinan si el operando es un registro (bit de modo = 0) o una posición de memoria (bit de modo = 1). Uno de los operandos o los dos pueden estar en registros, pero no pueden ser simultáneamente posiciones de memoria. Si el operando origen o el de destino corresponden a una posición de memoria, no se utilizan los campos de origen o de destino de la instrucción, dado que en su lugar se utiliza el campo de direcciones.

a. Escribir un programa que utilice código objeto (no simbólico) para intercambiar los contenidos de los registros R0 y R1. De ser necesario, pueden utilizarse los otros registros, pero no se debe emplear posición alguna de memoria. Usar no más de cuatro líneas de código (pueden ser menos). Colocar ceros en cualquier posición cuyo valor no interese.

b. Usar código objeto para escribir un programa que intercambie los contenidos de las posiciones de memoria 12 y 13. Como en la consigna anterior, está permitido utilizar los registros que sean necesarios, pero no posiciones de memoria. Nuevamente, colocar ceros en cualquier posición cuyo valor no interese.

4.7 Un programa ARC invoca la subrutina `foo`, a la que le transfiere tres argumentos, `a`, `b` y `c`. La subrutina tiene dos variables locales, `m` y `n`. Mostrar la posición del puntero de pila y los con-

FORMATO DE INSTRUCCIÓN



PATRONES BINARIOS

Modo	Patrón binario
Registro	0
Directo	1

REGISTRO PATRONES BINARIOS

Registro	Patrón binario
R0	00
R1	01
R2	10
R3	11

CONJUNTO DE INSTRUCCIÓN

Mnemónico	Código de operación	Significado
LOAD	000	Destino ← origen o memoria
STORE	001	Destino o memoria ← origen
ADD	010	Destino ← origen más destino
AND	011	Destino ← (AND) origen, destino
BZERO	100	Saltar si origen = 0
JUMP	101	Saltar incondicional
COMP	110	Destino ← complemento de origen
RSHIFT	111	Destino ← origen desplazado un lugar a derecha

Figura 4.27 • Arquitectura de programación de un sistema buscador de bolsillo.

tenidos de los elementos relevantes de la pila, cuando se invoca la subrutina mediante la técnica de utilización de la pila, en los puntos del programa que se indican. Nótese que la subrutina no devuelve valor alguno.

a. Justo antes del llamado, en el rótulo `x` antes de la instrucción `call`.
b. Cuando se completa la zona de pila para `foo`.
c. Justo antes de ejecutar la instrucción `ld`, en el rótulo `z` (o sea, cuando se reinicia la rutina que invoca a `foo`).

Utilizar la notación de pila de la figura 4.19.

```
! Empujar a la pila los argumentos a, b, c
x:  call  foo
z:  ld  %r1, %r2
        .
        .
        .
foo:! Aquí comienza la subrutina
        .
        .
        .
y:  jmpl  %r15 + 4, %r0
```

4.8 La instrucción `sethi` carga solo los 22 bits más significativos de un registro. ¿Por qué? Sería mucho más útil que `sethi` cargara los 32 bits del registro completo. ¿Cuál es el problema que surge si `sethi` carga la totalidad de los 32 bits?

4.9 ¿Cuál de las tres convenciones para llamado a subrutina analizadas en este capítulo es la que se utiliza en la figura 4.14?

4.10 Un programa compilado para la arquitectura de programación SPARC escribe el entero no signado de 32 bits 0xABCDEF01 en un archivo y lo vuelve a leer correctamente. El mismo programa compilado para Pentium también funciona en forma correcta. Sin embargo, cuando el archivo se transfiere entre máquinas, el programa lee incorrectamente el entero desde el archivo como 0x01EFCDAB. ¿Cuál es el problema?

4.11 Con referencia a la figura 4.25, indicar las instrucciones en lenguaje simbólico Java para el código que se ubica entre las direcciones `0x011e-0x0122`. Utilizar el formato sintáctico indicado en las direcciones `0x00e3-0x00ef` de la misma figura.
Será necesario hacer uso de las siguientes instrucciones Java:
`invokespecial n` (código de operación `0xb7`): invoca un método con índice `n` en el conjunto de constantes. Debe notarse que `n` es un índice de 16 bits que viene a continuacion del código de operación `invokespecial`.
`aload_0` (código de operación `0x2a`): colocar la variable local 0 en la pila.

4.12 ¿El formato de JVM para la representación de números es *big endian* o *little endian*? Ayuda: examinar la primera línea del programa bytecode de la figura 4.24.

4.13 Escribir un programa para ARC que implemente el programa *bytecode* de la figura 4.26. Suponer que, en forma similar a la del código de la figura, los argumentos se transfieren dentro de la pila y que el valor que se retorna vuelve dentro de la cabeza de la pila.

4.14 Suponer una implementación circuital de JVM, que pueda ejecutar los *bytecodes* de Java directamente en hardware de 32 bits.

a. ¿Cuánto tráfico en bytes se genera cuando se ejecuta el programa de la figura 4.26?

b. Resolver el problema 4.13 y calcular el tráfico de memoria generado por la ejecución de dicho programa. Comparar ese tráfico con la cantidad generada por el programa planteado en el ítem a del presente problema. Si la mayor parte del tiempo de ejecución de un programa se debe a sus accesos a memoria, ¿cuánto más rápido funcionará el programa aquí planteado con relación al de la figura 4.26?

4.15 ¿Puede un programa *bytecode* Java funcionar a la misma velocidad que un programa escrito en el lenguaje nativo del procesador? Justificar la respuesta en no más de dos párrafos.

4.16 a. Escribir programas en arquitecturas de tres direcciones, dos direcciones y una dirección que puedan calcular la función A = (B – C) * (D – E). Suponer códigos de operación de 8 bits, operandos y direcciones de 16 bits y transferencias de datos hacia y desde memoria efectuadas en conjuntos de 16 bits. Suponer, asimismo, que el código de operación debe transferirse desde la memoria por sí mismo. El código no deberá sobrescribir ninguno de los operandos. Podrán utilizarse todos los registros temporarios que fuesen necesarios.

b. Calcular el tamaño del programa medido en bytes.

c. Calcular el tráfico de memoria que generará el programa al momento de la ejecución, incluyendo las búsquedas de las instrucciones.

4.17 Repetir el problema 4.16, pero utilizando el lenguaje simbólico de ARC. Nótese que el mnemónico para la operación de resta es `subcc`, en tanto que el de la multiplicación es `smul`.

Capítulo 5

Los lenguajes y la máquina

En el capítulo anterior se analizaron las relaciones entre la arquitectura de programación, el lenguaje ensamblador y el lenguaje de máquina. También se analizaron con cierto detalle el procedimiento utilizado por las instrucciones para realizar transferencias entre registros y el movimiento de datos entre la memoria y la unidad central de proceso. Sin embargo, solo se trataron en forma breve los conceptos relacionados con el procedimiento de ensamblado y con el enlace (*linking*) y la carga de programas. En este capítulo se pretende ampliar la visión de las relaciones entre los lenguajes de computación y la máquina.

El análisis comienza por la **compilación**, definida como el proceso de traducción de un programa escrito en lenguaje de alto nivel a otro, funcionalmente equivalente, expresado en lenguaje ensamblador. A continuación se trata el proceso de **ensamblado** (*assembly*), definido como la traducción del programa escrito en lenguaje ensamblador a otro, funcionalmente equivalente, expresado en lenguaje de máquina. El análisis continúa con los procesos de **enlace** (*linking*), consistente en unir en un programa único distintos módulos que fueran ensamblados en forma separada, y de **carga**, lo que implica el traslado del programa a memoria y su preparación para ser ejecutado. Finalmente, se analiza el uso de **macroinstrucciones** del lenguaje ensamblador, las que pueden considerarse como elementos similares a los procedimientos en el momento del ensamblado, con la diferencia de que se expanden, en el programa simbólico, en cada una de las posiciones en las que se las invoca.

5.1 El proceso de compilación

Como se verá en secciones posteriores de este capítulo, el proceso de traducir un programa escrito en lenguaje ensamblador (simbólico) a lenguaje de máquina es muy directo, debido a que hay una relación uno a uno entre las sentencias del lenguaje simbólico y los códigos de máquina equivalentes. Los lenguajes de alto nivel, por el contrario, presentan un problema mucho más complejo.

5.1.1 Los pasos de la compilación

Considérese una simple sentencia de asignación, por ejemplo:

```
A = B + 4;
```

La conversión de esta sentencia en una o más sentencias de lenguaje simbólico enfrenta al compilador con una cantidad de tareas relativamente complejas:

- Reconocer dentro del texto del programa los símbolos básicos del lenguaje, por ejemplo, los identificadores tales como `A` y `B`, definiciones tales como la del valor constante `4` y delimitadores como = y +. Esta parte de la compilación se suele conocer como **análisis lexicográfico**.
- Analizar los símbolos para reconocer la estructura de programación subyacente. En el ejemplo, el **analizador** (*parser*) debe reconocer que la sentencia utilizada corresponde a una sentencia de asignación del tipo:

  Identificador "=" Expresión

  en la que, posteriormente, se analizará Expresión con el objeto de detectar una forma del tipo:

  Identificador "+" Constante

  Este procedimiento suele denominarse **análisis sintáctico**.

- Análisis de nombres. Asociar los nombres `A` y `B` con variables particulares del programa y, posteriormente, con posiciones particulares de memoria en las que se almacenarán dichas variables al momento de la ejecución del programa.
- Análisis de tipo. Determinar el tipo de todos los datos requeridos. En el ejemplo anterior, las variables `A` y `B` y la constante `4` se reconocerían, en ciertos lenguajes, como de tipo entero. Los análisis de nombre y de tipo se relacionan a menudo para incluirlos bajo el nombre de **análisis semántico**.
- **Asignación de acciones y generación de código**. Es la acción de asociar las sentencias de programa con la secuencia apropiada del lenguaje ensamblador. En el ejemplo de la sentencia anterior, la secuencia de lenguaje simbólico podría ser la siguiente:

```
                    ! Sentencia de asignación simple
ld [B], %r0, %r1    ! cargar la variable B en un registro
add %r1, 4 ,%r2     ! calcular el valor de la expresión
st %r2, %r0, [A]    ! asignar
```

- Además de las mencionadas, existen acciones adicionales que deben ser resueltas por el compilador, como la asignación de variables a registros, el control del uso de registros y, si así lo quisiera el programador, la optimización del programa.

5.1.2 La especificación del mapeo

Cuando se desarrolla el compilador, se debe incluir en su estructura la información acerca de la arquitectura de programación particular del procesador para el cual se lo desarrolla. (Nótese que la arquitectura de programación sobre la que el compilador se ejecuta no necesita ser igual a la del código que el compilador genera, proceso conocido como **compilación cruzada** [*cross compilation*].) Esta inclusión suele denominarse **especificación de asignación** del compilador. Por ejemplo, quien escribe compiladores debe decidir cómo ubicar los distintos tipos de variables y constantes dentro de los recursos de la máquina. Esto puede ser función tanto de la máquina como del lenguaje de alto nivel. En el lenguaje C de programación, por ejemplo, los valores enteros pueden ser de 16 bits, de 32 bits o de algún otro tamaño. Mientras tanto, Java especifica que el tamaño de todas las variables enteras debe ser de 32 bits. Para el ejemplo de la sección anterior, aplicado al lenguaje C, en ARC los enteros serán convertidos en palabras de 32 bits.

El autor del compilador debe tener en cuenta también las características y limitaciones de la máquina cuando convierte construcciones desde el lenguaje de alto nivel a sentencias del lenguaje simbólico o a secuencias de sentencias. Por ejemplo, el conjunto de instrucciones de ARC requiere que todos los operandos aritméticos sean constantes inmediatas o variables asignadas a registros. Por lo tanto, el compilador debe generar el código necesario para acomodar todas las variables en registros antes de poder ejecutar cualquier instrucción aritmética. Eso justifica, en el ejemplo anterior, la instrucción

```
ld [B], % r1
```

Este texto se centra en el análisis de los métodos que permiten convertir las construcciones generadas en un lenguaje común de alto nivel en construcciones equivalentes en el lenguaje simbólico. Los detalles referidos a los análisis lexicográfico, sintáctico y semántico se reservan para los textos que tratan específicamente el tema de compiladores. (En la sección "Para lectura posterior", que se encuentra al final del capítulo, se mencionan diversos textos sobre compiladores para aquel lector que tenga interés especial en el tema.)

5.1.3 Cómo convierte el compilador los tres tipos de instrucciones al código ensamblador

Se analizará en detalle la forma en que se lleva a cabo la conversión de los tres tipos básicos de instrucciones –para el movimiento de datos, aritméticas y de control– desde el lenguaje de alto nivel hacia el lenguaje ensamblador. Tanto para el análisis como para los

ejemplos incluidos se utilizará el lenguaje C como lenguaje de programación. Se adopta C debido a su popularidad y también a sus sintaxis y semántica, las que, si bien son de alto nivel, tienen cierta cercanía con los conceptos de los lenguajes de bajo nivel. El lector que no conozca el lenguaje C no debería preocuparse por esta elección, dado que la sintaxis y la semántica de C son sencillas de entender y de trasladar a otros lenguajes de alto nivel.

Almacenamiento de variables en memoria

En el ejemplo precedente, así como en la mayor parte de los ejemplos incluidos en este texto, se ha supuesto que se puede tener acceso a las variables directamente a través de su nombre, el que se asigna a una posición de memoria determinada en el momento de la traducción del programa simbólico, conocido como "tiempo de ensamblado" (*assembly time*). En el ejemplo anterior, `A = B + 4`, se supone que las variables `A` y `B` tienen direcciones que se conocen recién en el momento en que se realiza la compilación. De hecho, solo las **variables globales**, denominadas en C como **variables estáticas**, tienen direcciones conocidas en el momento de la compilación. Las variables declaradas dentro de funciones o dentro de bloques que no sean explícitamente declaradas como estáticas o globales, solo toman existencia cuando se ingresa en dicho bloque o función, desapareciendo cuando la función o bloque se invoca por última vez. Estas variables se conocen como **variables locales** o, en el caso de C, como **variables automáticas**. En la mayoría de los programas, las variables locales se emplean con más frecuencia que las variables globales.

Dada la naturaleza efímera de las variables locales, una forma natural de implementarlas es a través del uso de una pila tipo LIFO, de las que se han descripto en el capítulo 4. Las variables que se almacenan en la pila cobran valor cuando se crea la pila y se invoca la función, en tanto que desaparecen cuando se sale por última vez de dicha función. Si bien en el capítulo anterior se utilizó el registro puntero de pila `%sp` para el acceso a la pila, es habitual copiar el contenido de `%sp` a otro registro, llamado puntero base `%fp`, el que se utiliza para el acceso a las variables de la pila durante la vida de la función. Esto se justifica debido a que durante la ejecución de la función suele utilizarse la pila para el almacenamiento de variables temporarias, las que continuamente se ingresan y se sacan de la pila. Como consecuencia, se produce un desplazamiento variable entre `%sp` y los elementos almacenados en la pila. El uso de `%fp` implica que el compilador puede determinar un desplazamiento constante entre `%fp` y un valor dado almacenado en la pila, desplazamiento que se mantendrá fijo a lo largo del proceso. Para acceder a las variables de la pila se utiliza el **direccionamiento base**. Por ejemplo, una variable en ARC ubicada en la pila en una posición que se encuentre 12 bytes por debajo del valor de `%fp` puede cargarse en el registro `%r1` por medio de la instrucción

```
ld %fp, -12, %r1
```

O bien, utilizando una notación más común,

```
ld [%fp - 12], %r1
```

El uso del direccionamiento base permite entonces que el cálculo de direcciones del tipo "sumar el contenido del registro `%fp` con `-12`" se lleve a cabo en una sola instrucción. El direccionamiento base es bastante común como para que cualquier conjunto de instrucciones lo incluya dentro de sus direccionamientos. Algunos conjuntos de instrucciones contienen modos de direccionamiento aún más complicados, los que se utilizan cuando es necesario el acceso a estructuras de datos complejas almacenadas en el marco de la pila.

Para insistir sobre este punto, las variables que se almacenan en la pila están asignadas a direcciones que no se conocen hasta el momento de la ejecución. Sus direcciones en el momento de la compilación se conocen solo como desplazamientos respecto de `%fp`. La dirección real de memoria asignada a la variable solo se determina en el momento en que se ingresa en la función. No obstante, aun cuando sea mucho más común el uso de direcciones variables de pila como `[%fp - 12]` que el de direcciones globales como `A`, el análisis que sigue considerará el uso de variables globales debido a que es mucho más sencillo entender la relación entre el nombre de la variable, definido en el lenguaje de alto nivel, y su dirección especificada en el lenguaje de máquina. Con esta salvedad, se analizarán a continuación tres tipos de sentencias habituales en los programas: instrucciones para movimiento de datos, aritméticas y de control de secuencia.

5.1.4 Movimiento de datos

Además de las variables escalares más simples, la mayoría de los lenguajes de programación proveen diversos tipos de estructuras de datos complejas, incluyendo estructuras de registro fijo, como el tipo de datos `struct` del lenguaje C, similar a la estructura ***record*** de Pascal, y el tipo de dato *array*, común a la mayoría de los lenguajes de programación.

Estructuras

Para ejemplificar el uso de `struct` en C, puede considerarse la representación de un punto definido en un espacio tridimensional por tres coordenadas enteras x, y, z. En C, esta estructura se declara como

```
struct point {
   int x;
   int y;
   int z;
}
```

Una alternativa a esta `struct` se podría definir a través de la siguiente sentencia C:

```
struct point pt;
```

Habiendo definido la estructura correspondiente a `pt`, el programador puede referirse en forma individual a cada uno de los componentes de `pt`, a través de notaciones tales como `pt.x`, que se refiere al componente `x` de la estructura `pt`. El compilador acomodaría esta estructura en memoria en la forma de tres posiciones de memoria consecutivas.

La dirección de memoria de la estructura completa se toma sobre la base de la menor de todas, o dirección de base de la estructura, de modo que la componente `x` se almacenaría en la dirección `pt`, la componente `y` en `pt + 4`, y la componente `z` en `pt + 8`. Por lo tanto, la componente `y` de `pt` se podría cargar en el registro `%r1` por medio de la instrucción

```
ld [pt + 4], %r1        !%r1 ← y
```

Arreglos

La mayor parte de los lenguajes de programación permite también la declaración de arreglos de objetos, esto es, un conjunto de componentes idénticos a los que se puede hacer referencia tanto en forma individual como colectiva. En C, un arreglo de 10 enteros se puede definir como

```
int A[10];
```

Esta definición dará como resultado un conjunto de 10 enteros, indexados de 0 a 9.

Los componentes de una `struct` deben nombrarse en forma explícita durante la programación, por ejemplo, `pt.z`. No se admiten las referencias del tipo `pt.i` en las que `i` es una variable cuyo valor se calcula en el momento de la ejecución. En el caso de los arreglos, por otra parte, el índice del arreglo sí puede calcularse en el momento de la ejecución. Por ejemplo, el programador puede especificar el elemento `A[i]` del arreglo, en que `i` es una variable cuyo valor se determina en el momento de la ejecución, el que puede tomar cualquier valor entero entre 0 y 9. Mientras que en C el índice del primer elemento del arreglo siempre asume el valor de 0, otros lenguajes de programación admiten mayor flexibilidad. Por ejemplo, en Pascal, se permiten declaraciones de arreglos del tipo

```
A : array [-10 .. 10] of integer
```

Este tipo de declaración da como resultado un arreglo de 21 elementos cuyos índices van desde -10 hasta +10.

El acceso a los elementos del arreglo ofrece una dificultad algo mayor debido a la mencionada necesidad de calcular el índice en el momento de la ejecución y a la posibilidad de que los índices no se inicien en 0. La expresión general que permite calcular la dirección de memoria de un elemento de un arreglo en el momento de la ejecución está dada por

Dirección del Elemento = BASE + (ÍNDICE – COMIENZO) • TAMAÑO

Expresión en la que BASE es la dirección de comienzo del arreglo, ÍNDICE es el índice del elemento deseado, COMIENZO es el índice inicial del arreglo y TAMAÑO es el tamaño del elemento individual medido en bytes. Así, el elemento 5 del arreglo declarado anteriormente debería tener una dirección de A + (5 – (-10)) • 4 = A + 60.

En el lenguaje simbólico de ARC, suponiendo BASE en `%r2`, ÍNDICE en `%r3`, COMIENZO en `%r4` y TAMAÑO = 4, el código necesario para la carga de un elemento del arreglo en memoria sería:

```
sub %r3, %r4, %r6          ! %r6 ← ÍNDICE – COMIENZO
sll %r6, 2, %r6            ! %r6 ← %r6 * 4
ld [A + %r6], %r1          ! %r1 ← valor del arreglo
```

En esta expresión, `sll` es la instrucción de desplazamiento lógico a izquierda (*shift left logical*), que desplaza el registro `%r6` en dos bits a la izquierda, para permitir el acceso de dos ceros por la derecha del registro. Nótese que el acceso a un elemento del arreglo cuesta tres instrucciones, las que pueden ser más si TAMAÑO no es potencia de dos. Nótese, asimismo, que en el lenguaje de programación C, que especifica COMIENZO = 0, se ahorra una instrucción del lenguaje de máquina con cada acceso al arreglo. Esto produce una notable economía en los cálculos científicos y de ingeniería, en los que son habituales las referencias a arreglos.

5.1.5 Instrucciones aritméticas

Las instrucciones aritméticas se implementan en forma bastante similar a lo que se espera dada su utilización. No obstante, en aquellas máquinas que utilizan estructura de carga/descarga, como ARC y otras máquinas comerciales RISC, existen algunas posibles complicaciones. Independientemente de la cantidad de registros que la estructura de la máquina ofrezca, siempre es posible que el compilador encuentre alguna instrucción aritmética que requiera mayor cantidad de registros que la disponible. En este caso, el compilador deberá almacenar temporariamente algunas variables en la pila, debido a que se "derramaron" los registros. Los compiladores utilizan técnicas sofisticadas para determinar qué registros se encuentran disponibles, como la técnica gráfico-teórica, conocida como técnica de coloreo de registros, y para decidir cuándo el valor contenido en un registro deja de ser requerido para el almacenamiento de un valor particular, en un análisis conocido como de "vivo o muerto".

5.1.6 Control de secuencia

Muchas arquitecturas de programación utilizan bifurcaciones condicionales e incondicionales, así como las banderas aritméticas de la CPU para la implementación de sus estructuras de control de secuencia. En esta sección se analiza la forma de implementación de las sentencias de control más comunes.

La sentencia `goto`

La sentencia de control más trivial es la sentencia `goto`, del tipo `goto rótulo`, la que a nivel de lenguaje de máquina se implementa con una instrucción de salto incondicional `ba` (*branch always* [saltar siempre]):

```
ba    rótulo
```

La sentencia `if-else`

La sentencia `if-else` del lenguaje C tiene una sintaxis del tipo

```
if (expr) sentencia1 else sentencia2
```

la que se interpreta como "si la expresión considerada toma un valor verdadero, ejecutar la sentencia 1, en caso contrario ejecutar la sentencia 2". Por lo tanto, el compilador debe evaluar el valor de verdad de `expr` y ejecutar una de dos sentencias según la verdad o falsedad de la expresión `expr`. Supóngase, por razones de brevedad, que en el ejemplo siguiente la expresión a evaluar es (`%r1 == %r2`). La introducción de la instrucción salto por distinto, `bne` (*branch if not equal*), permite obtener este código para la implementación de la sentencia `if-else`:

```
subcc  %r1, %r2, %r0 ! fijar banderas, descartar resultado
bne  Over
! aquí va el código de la sentencia 1
ba End            ! salida del if-else
Over              ! aquí va el código de la sentencia 2
End               ! . . .
```

Nótese que el signo del salto condicional `bne`, salto por distinto, corresponde a la inversa de la expresión (`%r1 == %r2`), en la que se busca la igualdad de los dos registros. Esto se plantea así para que el programa en lenguaje de máquina vaya directamente a la sentencia 1 si se cumple la condición evaluada, en tanto que produce la bifurcación, saltando por encima de esta sentencia, si la condición no se cumple.

La sentencia `while`

La sentencia `while` en el lenguaje de programación C tiene como sintaxis

```
while (expresión) sentencia;
```

Esta sentencia tiene como significado: "evaluar la expresión; si es cierta, ejecutar la sentencia y repetir el proceso hasta que la expresión evaluada se convierta en falsa". La traducción de esta sentencia al lenguaje de máquina tiene como característica interesante el hecho de que la conversión más eficiente presenta la evaluación de la expresión después del código de la sentencia. Considérese la siguiente sentencia `while` en el lenguaje C:

```
while (%r1 == %r2) %r3 = %r3 + 1;
```

donde, nuevamente, se utilizan variables en registros para simplificar el código. La sentencia del lenguaje de alto nivel se implementa en forma eficiente por medio de

```
        ba Test
True:   add %r3, 1, %r3
Test:   subcc %r1, %r2, %r0
        be True
```

El lector podrá verificar que el hecho de colocar el código correspondiente a la evaluación de la expresión por debajo del código de la sentencia resulta más eficiente que si se tiene la evaluación antes del código de la sentencia.

La sentencia `do-while`

El lenguaje C tiene una sentencia `do-while`, cuya sintaxis es

```
do sentencia while (expresión);
```

`do-while` trabaja en forma similar a la sentencia `while`, excepto por el hecho de que la sentencia siempre se ejecuta una vez antes de evaluar la expresión. Se implementa exactamente de la misma forma que `while`, excepto por la eliminación de la primera instrucción `ba`.

La sentencia `for`

La sentencia `for` del lenguaje C tiene como sintaxis

```
for (expr1; expr2; expr3) sentencia;
```

La definición de esta sentencia en el lenguaje C dice que la misma equivale a

```
expr1;
while (expr2){
    sentencias
    expr3;
}
```

Por lo tanto, se implementa de la misma manera que las sentencias `while` anteriores, pero se agrega el código requerido para las expresiones `expr1` y `expr3`.

5.2 El proceso de ensamblado

El proceso de traducción de un programa expresado en lenguaje simbólico se conoce como **proceso de ensamblado**. Este proceso es lineal y relativamente simple, dado que hay una relación directa y unívoca entre las sentencias del lenguaje simbólico y sus correspondientes expresiones en el lenguaje de máquina. Esto marca una diferencia con el proceso de compilación, por ejemplo, en el que una sentencia de un lenguaje de alto nivel dado puede traducirse como una buena cantidad de sentencias en el lenguaje de máquina.

Si bien el ensamblado es un proceso lineal, es tedioso y proclive al error si se hace a mano. De hecho, el programa ensamblador es una de las primeras herramientas de software desarrolladas luego de la invención de la computadora electrónica digital.

Los ensambladores comerciales deben ofrecer, al menos, las siguientes prestaciones:

- Permitir que el programador especifique la ubicación de las variables y programas al momento de la ejecución. (Sin embargo, en la mayoría de los casos el programador no especifica una dirección absoluta de comienzo de un programa, dado que el programa puede ser movido o reubicado por el programa vinculador (*linker*) y, probablemente, por el programa de carga (*loader*), de acuerdo con lo que se analizará más adelante.)
- Ofrecerle al programador la posibilidad de inicializar los valores de los datos en memoria antes de la ejecución del programa.
- Proveer expresiones nemotécnicas en el lenguaje de programación para todas las instrucciones del lenguaje de máquina y modos de direccionamiento, y traducir las

sentencias válidas del lenguaje simbólico hacia sus valores binarios equivalentes en el lenguaje absoluto.
- Permitir el uso de rótulos simbólicos para identificar o representar direcciones y constantes.
- Ofrecerle al programador una forma de especificar la dirección de comienzo de un programa, si existiera. (Por ejemplo, no habría dirección de comienzo si el módulo que se traduce es una función o un procedimiento.)
- Ofrecer cierto grado de cálculo al momento del ensamblado.
- Incluir un mecanismo que permita la definición de variables en un programa escrito en lenguaje simbólico y el uso de las mismas en otro programa ensamblado por separado.
- Proveer la expansión de **macro rutinas**, o sea, rutinas que puedan definirse una vez y utilizarse tantas veces como sea necesario.

A continuación se ilustra el funcionamiento del proceso de ensamblado a través del proceso de "ensamblado a mano" de un programa sencillo, expresado en lenguaje simbólico de ARC hacia su lenguaje de máquina. El programa a ensamblar es similar al de la figura 4.13, que se reproduce en la figura 5.1 por razones de comodidad. Al ensamblar el programa se utilizan los formatos de codificación para ARC que fueran presentados en la figura 4.10, la que se reproduce aquí como figura 5.2. La figura muestra la codificación del lenguaje de máquina de ARC. Esto es, especifica el lenguaje de máquina binario que el ensamblador debe generar a partir del texto del lenguaje de programación.

```
! Este programa suma dos números
        .begin
        .org 2048
main:   ld      [x], %r1          ! Cargar x en el registro %r1
        ld      [y], %r2          ! Cargar y en el registro %r2
        addcc   %r1, %r2, %r3     ! %r3 ← %r1 + %r2
        st      %r3, [z]          ! Almacenar el registro %r3 en z
        jmpl    %r15 + 4, %r0     ! Retorno
x:      15
y:      9
z:      0
        .end
```

Figura 5.1 • Un programa simple para ARC que suma dos números.

5.2.1 El proceso de ensamblado y los ensambladores de dos pasadas

La mayoría de los ensambladores recorren dos veces el texto escrito en lenguaje simbólico, por lo que se los conoce como "ensambladores de dos pasadas". En la primera pasada el ensamblador se dedica a determinar las direcciones de todos los datos e instrucciones del programa y a seleccionar qué instrucción del lenguaje de máquina debe generarse para cada instrucción del lenguaje simbólico, pero sin generar aún el código de máquina.

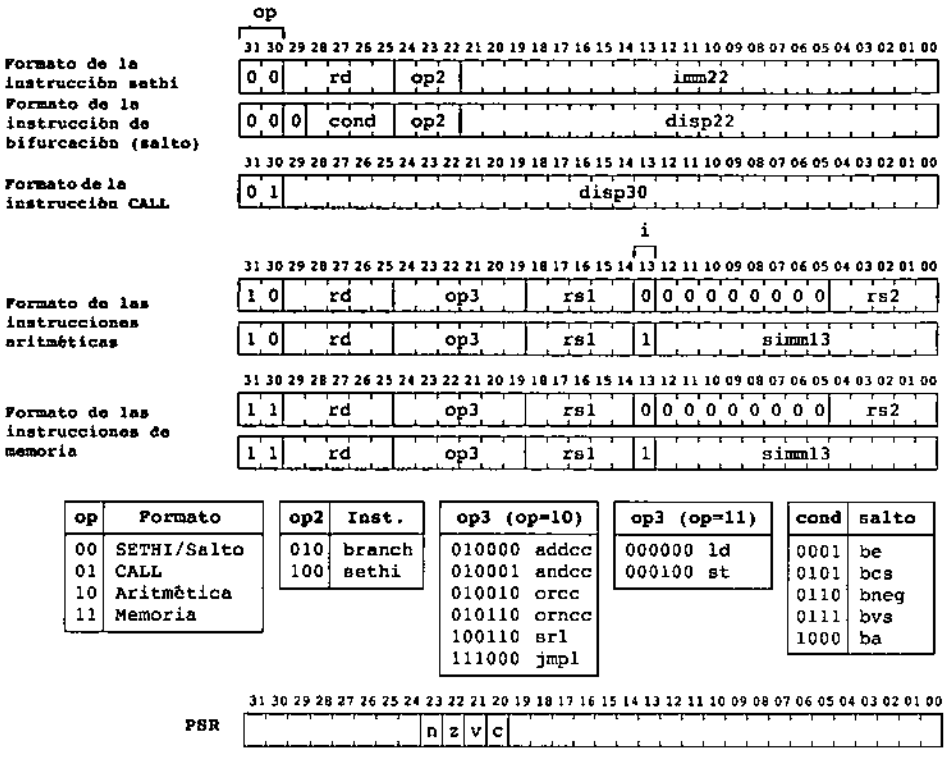


Figura 5.2 • Formatos de instrucción y formatos PSR para ARC.

Las direcciones de los datos y de las instrucciones se determinan en el momento del ensamblado mediante un contador similar al contador de programa, al que se conoce como **contador de posición** (*location counter*). A medida que se desarrolla el proceso de ensamblado, este contador de posición lleva el control de la dirección de la instrucción o del elemento de datos corriente. Generalmente, se lo inicializa a 0 al comienzo del primer paso y se lo incrementa en pasos equivalentes al tamaño de cada instrucción. La directiva `.org` hace que el contador se ubique en el valor especificado por la misma. Por ejemplo, en caso de encontrar la sentencia

```
.org 1000
```

el contador de posición adoptará el valor 1000, por lo que la instrucción o dato siguiente se asignará a dicha dirección. Durante esta pasada, el ensamblador realiza también cualquier operación aritmética necesaria e inserta las definiciones de todos los rótulos y constantes en una **tabla de símbolos**.

La razón principal para requerir una segunda pasada tiene que ver con la necesidad de permitir el uso de los símbolos dentro de un programa con anterioridad a que sean definidos, lo que se suele conocer como **referencia previa**. Luego de la primera pasada, el ensamblador tendrá todos sus símbolos definidos e ingresados en la tabla de símbolos,

por lo que durante la segunda pasada podrá generar el código de máquina, insertando en el mismo los valores de los símbolos, ya conocidos para ese momento.

A continuación se procederá a efectuar la traducción manual del programa de la figura 5.1 al lenguaje de máquina. Cuando el ensamblador encuentra la primera instrucción,

```
ld      [x], %r1
```

utiliza un proceso de verificación de coincidencia de patrones de texto para reconocer que se trata de una instrucción de carga. De un análisis posterior deduce que el mismo tiene la forma "cargar desde una dirección de memoria especificada como un valor constante (en este caso `x`) más el contenido de un registro (en este caso `%r0`) hacia un registro (`%r1` en el ejemplo)". Esta expresión corresponde al segundo de los formatos de memoria representados en la figura 5.2. Al examinar el segundo formato de memoria se puede observar que el campo de operación `op` para la instrucción (`ld`) es `11`. El destino de esta instrucción `ld` se indica en el campo `rd`, que en este caso es `00001` por el registro `%r1`. El campo `op3` para la operación `ld` es `00000`, tal como se indica en el cuadro `op3` debajo de los formatos de memoria. El campo `rs1` identifica el registro, `%r0` en este caso, que se suma al campo `simm13` para formar la dirección del operando origen. A continuación, viene el bit `i`. Nótese que este bit se utiliza para distinguir el primer formato de memoria (`i=0`) del segundo (`i=1`). Por consiguiente, en este caso, el bit `i` adopta el valor 1. El campo `simm13` especifica la dirección del rótulo `x`, que aparece cinco palabras después de la primera instrucción. Dado que la primera instrucción se plantea en la dirección 2048 y, además, que cada palabra está compuesta por cuatro bytes, la dirección de `x` se encuentra 5 x 4 = 20 bytes después del comienzo del programa. La dirección de `x` es, por lo tanto, 2048 + 20 = 2068, representado por la palabra binaria 0 1000 0001 0100. Este patrón cabe dentro del campo `simm13`, signado y de 13 bits.

La primera línea, por ende, se traduce en la siguiente combinación binaria:

```
11 00001 000000 00000 1 0100000010100
op   rd     op3    rs1  i      simm13
```

La próxima instrucción tiene un formato similar, siendo su patrón binario el que se muestra a continuación:

```
11 00010 000000 00000 1 0100000011000
op   rd     op3    rs1  i      simm13
```

El proceso de ensamblado continúa hasta que se logra la traducción de la totalidad de las ocho líneas, según se muestra a continuación:

```
ld [x], %r1          1100 0010 0000 0000 0010 1000 0001 0100
ld [y], %r2          1100 0100 0000 0000 0010 1000 0001 1000
addcc %r1, %r2,%r3   1000 0110 1000 0000 0100 0000 0000 0010
st %r3, [z]          1100 0110 0100 0000 0010 1000 0001 1100
jmpl %r15+4, %r0     1000 0001 1100 0011 1110 0000 0000 0100
15                   0000 0000 0000 0000 0000 0000 0000 1111
9                    0000 0000 0000 0000 0000 0000 0000 1001
0                    0000 0000 0000 0000 0000 0000 0000 0000
```

Como técnica general, el proceso de ensamblado se lleva a cabo leyendo las sentencias del lenguaje simbólico en forma secuencial, desde la primera hasta la última, y generando el código de máquina para cada una de las sentencias. Como se ha mencionado anteriormente, las referencias previas causan una dificultad importante en esta metodología. Si se considera el fragmento de programa de la figura 5.3, cuando el ensamblador encuentra la sentencia de `call` no sabe la ubicación de `sub_r`, debido a que aún no detectó el rótulo correspondiente. Por consiguiente, la referencia se ingresa en la tabla de símbolos y se señala como indeterminada. La referencia se resuelve posteriormente en la secuencia del programa, cuando el ensamblador detecta la definición de `sub_r`. Más adelante se describe el proceso de construcción de una tabla de símbolos.

```
                .
                .
                .
        call    sub_r       ! Punto en que se invoca la subrutina
                .
                .
                .
sub_r:  st      %r1, [w]    ! Punto en que se define la subrutina
                .
                .
                .
```

Figura 5.3 • Un ejemplo de referencia previa.

5.2.2 El proceso de ensamblado y la tabla de símbolos

En la primera pasada de un proceso de ensamblado de dos pasadas se crea una **tabla de símbolos**. Un símbolo puede ser un rótulo o un nombre simbólico que se refiera a un valor utilizado durante el proceso de ensamblado.

Como ejemplo del funcionamiento de un ensamblador de dos pasadas, considérese la traducción del código de la figura 4.14. Si se comienza a partir de la sentencia de `.begin`, el programa ensamblador encuentra la sentencia

```
.org 2048
```

Esta sentencia hace que el ensamblador coloque el contador de posiciones en 2048, con lo que el proceso de traducción se inicia en esa dirección. La primera sentencia encontrada es

```
a_start     .equ      3000
```

Se genera en la tabla de símbolos la entrada correspondiente a `a_start`, a la que se le asigna el valor 3000. (Debe notarse que las sentencias `.equ` no generan código alguno, por lo que no se les asigna dirección durante el ensamblado.)

El proceso de traducción continúa y el traductor encuentra la primera instrucción del lenguaje

```
ld     [length],     %r1
```

Esta instrucción se traduce en la dirección 2048 especificada por el contador de posiciones. Se procede a incrementar el contador en un valor de 4, equivalente al tamaño en bytes de la instrucción, por lo que adopta el valor 2052. Nótese que cuando el ensamblador detecta el símbolo `length` no ha encontrado aún definición alguna del mismo. Se genera una entrada en la tabla de símbolos, asignada a la variable `length`, como indefinida, tal como lo muestra la figura 5.4a.

Símbolo	Valor
a_start	3000
length	—

(a)

Símbolo	Valor
a_start	3000
length	2096
address	2100
loop	2060
done	2088
a	3000

(b)

Figura 5.4 • Tabla de símbolos para el programa ARC de la figura 4.14, (a) luego de encontrar los símbolos `a_start` y `length`; y (b) al finalizar.

El ensamblador detecta luego la segunda instrucción:

```
ld     [address],     %r2
```

El programa traduce la instrucción en la dirección 2052 y genera una entrada para el símbolo `address` en la tabla de símbolos, fijando nuevamente su valor como indeterminado, dado que su definición aún no ha sido encontrada. Incrementa luego el contador de posiciones en 4 para llevarlo a 2056. La instrucción `andcc` se ensambla en la dirección 2056 y el contador de posiciones vuelve a incrementarse en el tamaño de la instrucción, nuevamente de 4 bytes, pa-

ra llevarlo a 2060. El símbolo detectado a continuación es `loop`, que se ingresa en la tabla de símbolos con un valor de 2060, el valor del contador de posiciones. El siguiente símbolo detectado que no se encuentra en la tabla de símbolos es `done`, que también se ingresa en la tabla de símbolos sin valor alguno, dado que tampoco ha sido definido aún.

Durante la primera pasada del ensamblador se van asignando las variables `length`, `address` y `done` con los valores numéricos 2092, 2096 y 2088, respectivamente, a medida que se los va encontrando. Cuando se detecta el rótulo `a`, se lo ingresa en la tabla asignándosele el valor 3000. El rótulo `done` aparece en la dirección 2088 debido a que hay 10 instrucciones (40 bytes) entre el comienzo del programa y el rótulo en cuestión. Las direcciones de los rótulos restantes se calculan en forma similar. Si al final de la primera pasada quedaran todavía rótulos sin definir, existe un error en el programa; en tal caso, el ensamblador señalará los símbolos que no fueron definidos y terminará.

Luego de creada la tabla de símbolos comienza la segunda vuelta del ensamblado. El programa se lee por segunda vez, empezando desde la sentencia de `.begin`, para generar en esta vuelta el código objeto. La primera sentencia encontrada que genera código es la de `ld`, en la dirección 2048. La tabla de símbolos indica que el campo de direcciones de la instrucción `ld` es $(2092)_{10}$, correspondiente a la dirección de la variable `length`, por lo que se genera una palabra de código utilizando el formato correspondiente a una instrucción de memoria, tal como se muestra en la figura 5.5. La segunda pasada continúa de la misma forma hasta completar la traducción de todo el código. La figura 5.5 muestra el programa ensamblado. Nótese que el desplazamiento correspondiente a las instrucciones de bifurcación se da en palabras, y no en bytes, debido a que las instrucciones de bifurcación multiplican los desplazamientos en un factor de cuatro.

5.2.3 Tareas finales del programa ensamblador

Luego de completar el proceso de traducción, el ensamblador debe agregarle al módulo traducido información adicional para uso de los programas de enlace y carga:

- El nombre y tamaño del módulo. Si el modelo de ejecución involucra segmentos de memoria para código, datos, pila, etc., deben especificarse los tamaños e identidades de los distintos segmentos.
- La dirección del símbolo de comienzo, si es que se define alguno en el módulo. La mayoría de los ensambladores y lenguajes de alto nivel proveen un rótulo especial reservado que el programador puede utilizar para indicar cuál debe ser el lugar de comienzo de ejecución del programa. Por ejemplo, C especifica que la ejecución se debe iniciar en la función llamada `main()`. En la figura 5.1, el rótulo "main" le indica al ensamblador que esa debe ser la dirección de comienzo de la ejecución del programa.
- Información acerca de símbolos globales y externos. El programa de enlace necesitará conocer la dirección de cualquier símbolo global definido en el módulo y expor-

Contador de posición	Instrucción		Código objeto
	.begin		
	.org	2048	
a_start	.equ	3000	
2048	ld	[length],%r1	11000010 00000000 00101000 00101100
2052	ld	[address],%r2	11000100 00000000 00101000 00110000
2056	andcc	%r3,%r0,%r3	10000110 10001000 11000000 00000000
2060 loop:	andcc	%r1,%r1,%r0	10000000 10001000 01000000 00000001
2064	be	done	00000010 10000000 00000000 00000110
2068	addcc	%r1,-4,%r1	10000010 10000000 01111111 11111100
2072	addcc	%r1,%r2,%r4	10001000 10000000 01000000 00000010
2076	ld	%r4,%r5	11001010 00000001 00000000 00000000
2080	ba	loop	00010000 10111111 11111111 11111011
2084	addcc	%r3,%r5,%r3	10000110 10000000 11000000 00000101
2088 done:	jmpl	%r15+4,%r0	10000001 11000011 11100000 00000100
2092 length:		20	00000000 00000000 00000000 00010100
2096 address:		a_start	00000000 00000000 00001011 10111000
	.org	a_start	
3000 a:		25]	00000000 00000000 00000000 00011001
3004		-10]	11111111 11111111 11111111 11110110
3008		33	00000000 00000000 00000000 00100001
3012		-5]	11111111 11111111 11111111 11111011
3016		7]	00000000 00000000 00000000 00000111
	.end		

Figura 5.5 • Salida proveniente de la segunda pasada de ensamblado del programa ARC de la figura 4.14.

tado por el mismo, así como necesitará saber qué símbolos se encuentran indefinidos en el módulo porque se definen como globales en otro módulo.

- Información acerca de las rutinas de biblioteca a las que el módulo hace referencia. Algunas bibliotecas contienen elementos de uso común así como funciones matemáticas u otras funciones especiales. Hay mucho más para decir en próximas secciones acerca del uso de bibliotecas.
- Los valores de cualquier constante que deba cargarse en memoria. Algunos programas de carga esperan que la inicialización de los datos se especifique en forma separada del código binario.
- Información de reubicación. Cuando se invoca al programa de enlace, la mayoría de los módulos a vincular deben reubicarse a medida que se los concatena. Todo el procedimiento de reubicación de módulos es complicado debido a que algunas referencias a direcciones pueden reubicarse y otras no. El tema de la reubicación se analizará más adelante, pero aquí se debe hacer notar que el ensamblador puede especificar cuáles direcciones pueden reubicarse y cuáles no.

5.2.4 Ubicación de programas en memoria

Hasta ahora se ha supuesto que los programas se encuentran localizados en memoria, en una dirección especificada por una directiva `.org`. Esto puede ser cierto en algunos casos de programación de sistemas, cuando el programador tiene razones para querer que un programa se ubique en una posición específica de memoria. En la mayoría de los casos, al programador no le interesa en qué posición de la memoria se carga el programa. Más aún, cuando se vinculan entre sí programas que han sido compilados o ensamblados en forma separada, es difícil o imposible que el programador sepa exactamente en qué lugar terminará ubicado cada módulo luego del enlace de los mismos, a medida que se los coloca uno tras otro. Por esta causa, la mayoría de las direcciones se especifican como **reubicables** en memoria, excepto quizás por direcciones como las de entrada-salida, que pueden estar fijas en una dirección absoluta de memoria.

En la sección siguiente se analiza con más detalle el proceso de reubicación. Aquí, simplemente se menciona que es responsabilidad del ensamblador indicar qué símbolos son reubicables. La característica de reubicable o no que tiene un símbolo determinado depende tanto del lenguaje simbólico como de las convenciones del sistema operativo. En cualquier caso, la información de reubicación se incluye en un **diccionario de reubicación**, en el módulo ensamblado, para que sea utilizado por el editor de enlace (*linker*) y/o el cargador. Los símbolos reubicables se señalizan habitualmente con una "R" que aparece luego de su valor en los listados generados por el ensamblador.

5.3 Enlace y carga

La mayoría de las aplicaciones de cualquier tamaño tienen una cantidad de módulos que han sido compilados o ensamblados por separado. Estos módulos pueden haber sido generados por distintos lenguajes de programación o pueden integrar una biblioteca provista como parte del ambiente del lenguaje de programación o del sistema operativo. Cada módulo deberá proveer la información descripta anteriormente, de modo que todos ellos puedan vincularse en forma conjunta para su carga y ejecución.

Un **editor de enlace**, o ***linker***, es un programa que combina programas ensamblados por separado (llamados **módulos objeto**) en un único programa, o **módulo de carga**. El *linker* resuelve todas las referencias globales y externas y reubica las direcciones de los diferentes módulos. El módulo de carga puede ser cargado en memoria por medio de un **cargador**, que también puede necesitar modificar direcciones si el programa se carga en una dirección distinta a la dirección de origen de carga usada por el *linker*.

Una técnica relativamente nueva, denominada **bibliotecas de enlace dinámico** (DLL, *dynamic link libraries*), popularizada por Microsoft en su sistema operativo Windows y presente en formas similares en otros sistemas operativos, pospone el enlace de algunos componentes hasta que sean requeridos efectivamente durante el mo-

mento de la ejecución. En apartados posteriores de esta sección habrá más para comentar acerca de las vinculaciones dinámicas.

5.3.1 Enlace (linking)

Al combinar los módulos ensamblados o compilados por separado en un módulo de carga, el programa de enlace debe:

- Resolver referencias de direcciones externas a los módulos a medida que los vincula.
- Reubicar cada módulo por medio de la combinación más apropiada de los mismos. Durante este proceso de reubicación muchas de las direcciones del módulo pueden llegar a modificarse para reflejar su nueva ubicación.
- Especificar el símbolo de comienzo del módulo de carga.
- Si el modelo de memoria incluye más de un segmento de memoria, debe especificar los contenidos e identidades de los diversos segmentos.

Resolución de referencias externas

Al resolver las referencias de direcciones, el programa de enlace necesita distinguir los nombres de los símbolos locales (aquellos que se utilizan dentro de un único módulo fuente) de los nombres de los símbolos globales (los que se usan en más de un módulo). Esto se logra mediante el uso, durante el proceso de ensamblado, de las directivas `.global` y `.extern`. La directiva `.global` le indica al ensamblador que debe señalizar al símbolo como disponible para otros módulos objeto durante la fase de enlace. La directiva `.extern` identifica a un rótulo usado en un módulo pero que se encuentra definido en otro. Por lo tanto, se utiliza la directiva `.global` en el módulo en que se define un símbolo (como el módulo en el que se encuentra ubicada una subrutina) y se utiliza la directiva `.extern` en cada uno de los módulos que haga referencia a aquel. Nótese que solo los rótulos de direcciones pueden ser globales o externos. No tendría sentido definir como global o externo un símbolo `.equ`, dado que `.equ` es una directiva que solo se usa durante el ensamblado, el que se habrá completado en el momento que se inicia el proceso de enlace.

Todos aquellos rótulos de un programa a los que se haga referencia desde otros, como los nombres de las subrutinas, contarán en el módulo fuente con una línea de formato similar al siguiente:

```
.global símbolo1, símbolo2, ...
```

Todos los rótulos restantes son locales, lo que significa que se puede utilizar el mismo rótulo en más de un módulo fuente sin el riesgo de confundirlos, dado que los rótulos locales no se utilizan luego de finalizado el proceso de ensamblado. Un módulo que se refie-

re a símbolos definidos en otro modulo debería declarar estos símbolos usando una forma del tipo

```
.extern símbolo1, símbolo2, ....
```

Como ejemplo de cómo se utilizan las directivas `.global` y `.extern`, considérense los dos módulos de código simbólico de la figura 5.6. Cada uno de los módulos se ensambla, por separado, en un módulo objeto; cada uno con su propia tabla de símbolos, según lo ilustra la figura 5.7. Las tablas de símbolos tienen un campo adicional donde se indica si un símbolo dado es externo o global. El programa principal (`main`) comienza en la posición 2048, y dado que cada una de las instrucciones tiene una longitud de cuatro bytes, las variables `x` e `y` se encuentran, respectivamente, en las posiciones 2064 y 2068. El símbolo `sub` se encuentra indicado como externo a consecuencia de la directiva `.extern`. Como parte del proceso de traducción, el ensamblador incluye un encabezado en el módulo con información acerca de los símbolos que son globales o externos, para que puedan ser resueltos en el momento del enlace.

```
! Programa principal
      .begin
      .org  2048
      .extern sub
main: ld    [x], %r2
      ld    [y], %r3
      call  sub
      jmpl  %r15 + 4, %r0
   x: 105
   y:  92
      .end
```

```
! Biblioteca de subrutinas
      .begin
ONE   .equ     1
      .org   2048
      .global sub
sub:  orncc  %r3, %r0, %r3
      addcc  %r3, ONE, %r3
      jmpl   %r15 + 4, %r0
      .end
```

Figura 5.6 • Un programa que invoca una subrutina de resta de dos números enteros.

Símbolo	Valor	Global/ Externa	Reubicable
sub	–	Externa	–
main	2048	No	Sí
x	2064	No	Sí
y	2068	No	Sí

Programa principal

Símbolo	Valor	Global/ Externa	Reubicable
ONE	1	No	No
sub	2048	Global	Sí

Biblioteca de subrutinas

Figura 5.7 • Tablas de símbolos para los módulos de código fuente de la figura 5.6.

Reubicación

En la figura 5.6 puede observarse que los dos programas, `main` y `sub`, tienen la misma dirección de comienzo, 2048. Obviamente, no pueden ocupar ambos esa misma dirección de memoria. Si los dos módulos se ensamblan por separado, no hay forma de que un programa ensamblador descubra el conflicto entre las direcciones de memoria durante el proceso de traducción. Para resolver este problema, el ensamblador define como **reubicables** a aquellos símbolos que pueden admitir que sus direcciones sean modificadas durante el proceso de enlace, como se muestra en los campos correspondientes de las tablas de símbolos de la figura 5.7. La idea es que un programa ensamblado en una dirección inicial como 2048 pueda cargarse no a partir de ella sino a partir de otra dirección como, por ejemplo, 3000. Para permitir esta modificación, todas las referencias a direcciones reubicables incluidas en el programa deben incrementarse en 3000 – 2048 = 952. La reubicación se realiza a través del programa *linker*, de modo tal que las direcciones reubicables se modifican en el mismo valor en que se modificó la dirección de origen de carga. En cambio, las direcciones absolutas, no reubicables, (tales como la dirección más alta posible para la pila, que vale $2^{31} - 4$ para palabras de 32 bits), se mantienen sin modificación independientemente del origen de la carga.

El ensamblador tiene la responsabilidad de determinar qué rótulos son reubicables cuando construye la tabla de símbolos. No tiene sentido determinar como reubicable un rótulo externo, dado que el rótulo se define en otro módulo, por lo que `sub` no tiene entrada reubicable en la tabla de símbolos del programa `main`, en la figura 5.7, pero se señala como reubicable en la biblioteca de subrutinas. Asimismo, el ensamblador tiene que identificar los códigos del módulo objeto que deben ser modificados como consecuencia de la reubicación. Los números absolutos, por ejemplo las constantes (señaladas por `.equ`, o las que aparecen en posiciones de memoria, tales como los contenidos de `x` e `y`, que son 105 y 92, respectivamente) no son reubicables. Las posiciones de memoria ubicadas en forma relativa a una sentencia `.org`, tales como `x` e `y` (¡no los contenidos de `x` ni de `y`!) suelen ser reubicables. Las referencias a posiciones fijas, como podrían ser las de una rutina gráfica residente en forma permanente que puede estar eléctricamente instalada en la máquina, no son reubicables. Toda la información requerida para reubicar un módulo se almacena en el diccionario de reubicación contenido en el archivo ensamblado, por lo que se encuentra disponible para el programa *linker*.

5.3.2 Carga

El **cargador** (*loader*) es un programa que ubica el módulo de carga en la memoria principal. Conceptualmente, las tareas del programa cargador no son complejas. Debe cargar los diversos segmentos de memoria con los valores apropiados e inicializar ciertos registros, como el puntero de pila `%sp` y el contador de programa `%pc`, a sus valores iniciales.

Si en todo momento se tiene en ejecución un único modulo de carga, el modelo funciona correctamente. Sin embargo, en los sistemas operativos modernos, en todo momento existen varios programas que residen en memoria, y, por consiguiente, no hay forma de que el ensamblador o el *linker* conozcan cuál será la dirección en la que van a residir. El programa cargador debe reubicar estos módulos en el momento de la carga, para lo cual le sumará un desplazamiento a todo el código reubicable de un módulo dado. Este tipo de cargador se conoce como **cargador reubicador**. Este programa no repite el trabajo del *linker*. El programa de enlace debe combinar varios módulos objeto en un único módulo de carga, mientras que el cargador simplemente modifica las direcciones reubicables que encuentra dentro de un módulo de carga dado para permitir la coexistencia en memoria de varios programas en forma simultánea. Un programa **cargador y de enlace** (*linking loader*) ejecuta tanto el proceso de enlace como el de carga: resuelve referencias externas, reubica módulos objeto y los carga en memoria.

El archivo ejecutable contiene un encabezado con información que describe dónde se lo debe cargar, cuál es su dirección de comienzo y si es posible que incluya información de reubicación así como puntos de entrada para cualquier rutina que deba estar disponible hacia el exterior del mismo.

Una aproximación alternativa, que se basa en las técnicas de administración de memoria, realiza la reubicación por medio de la carga, en un registro base de segmento, de la base apropiada para la ubicación del código (o de los datos) en el correspondiente lugar de la memoria física. La **unidad de administración de memoria** (MMU, *memory management unit*) suma el contenido de este registro base a todas las referencias de memoria. Como resultado, cada programa puede iniciar su ejecución en la dirección 0 y confiar en la MMU para lograr la reubicación de todas las referencias de memoria en forma transparente.

Bibliotecas de enlace dinámico (DLL)

Volviendo a las bibliotecas de enlace dinámico, el concepto tiene una cantidad de prestaciones atractivas. Las rutinas usadas habitualmente, como los paquetes de software para manejo de memoria o de gráficos, solo deben estar presentes en un lugar, la biblioteca DLL. Esto da por resultado un programa de menor tamaño, dado que no hace falta que cada programa posea su propia copia del código DLL, como pasaría en otras situaciones. Todos los programas comparten exactamente el mismo código, aun cuando se ejecutan simultáneamente.

Más aún, la actualización de las bibliotecas DLL para incorporar mejoras en sus prestaciones o eliminar errores requiere que las correspondientes modificaciones sean incorporadas en un solo punto, con lo que los programas que las utilizan no necesitan ser recompilados en un paso separado. No obstante, estas mismas prestaciones se pueden convertir en desventajas porque la conducta del programa puede modificarse en forma no deseada (por ejemplo, cuando un programa se queda sin memoria por el uso de una bi-

blioteca DLL de mayor tamaño). Las bibliotecas DLL deben estar presentes en todo momento y tienen que contener la versión esperada por cada programa. Muchos usuarios del sistema operativo Windows deben haber visto el críptico mensaje "Falta un archivo en la biblioteca de enlace dinámico". Para complicar aún más el problema en la implementación de Windows, el sistema de archivos presenta diferentes lugares en donde almacenar bibliotecas DLL. Estos problemas pueden llegar a ser resueltos con poca dificultad por un usuario experto; en cambio, resultarán un problema insoluble para el usuario común.

Ejemplo de programación

Considérese el problema de sumar dos números de 64 bits cada uno utilizando el lenguaje de máquina del procesador ARC. Los dos números de 64 bits pueden almacenarse en posiciones consecutivas de memoria, sumando luego, por separado, las palabras de menor y mayor orden. Si se genera un arrastre en la suma de las palabras menos significativas, ese arrastre deberá sumarse con la palabra más significativa del resultado. (Véase el problema 5.3 para la generación de la tabla de símbolos y el problema 5.4 para la traducción del código simbólico a lenguaje de máquina.)

La figura 5.8 muestra una posible codificación. Los dos números de 64 bits `A` y `B` se almacenan en memoria en el formato *big-endian*, en el cual los 32 bits más significativos se almacenan en una dirección de memoria menor que la que almacena los 32 bits menos significativos. El programa comienza cargando las dos palabras de `A` en los registros `%r1` y `%r2`, que contienen, respectivamente, las palabras más y menos significativas de `A`. Luego, se cargan las palabras más y menos significativas de `B` en los registros `%r3` y `%r4`, respectivamente. El programa invoca la rutina `add_64`, cuya función es sumar `A` y `B` y almacenar la palabra más significativa del resultado en `%r5` y la menos significativa en `%r6`. El resultado, de 64 bits, se almacena en `C`, con lo que se completa la ejecución del programa.

La subrutina `add_64` comienza sumando las dos palabras de menor orden. Si no se produce arrastre, se suman a continuación las dos palabras de mayor orden y se completa la ejecución de la subrutina. Si se genera arrastre en la suma de las dos palabras menos significativas, este debe sumarse con la palabra de mayor orden del resultado. Si al sumar las palabras más significativas de los operandos no se produce arrastre, el paso planteado consiste, simplemente, en sumar el arrastre producido en las palabras de menor orden al resultado final. En cambio, si se produce un arrastre a partir de la suma de las dos palabras de mayor orden, el hecho de agregar al resultado el arrastre generado en la suma de las palabras menos significativas provocará un estado final de los códigos de condición en el que no se reflejará el arrastre generado en la mitad más significativa. El código de condición correspondiente al arrastre se puede regenerar colocando un número suficientemente grande en `%r7` y sumándolo consigo mismo. No obstante, este procedimiento puede dejar valores incorrectos en los códigos de condición correspondientes a `n`, `z` y `v`. La solución completa no se detalla en este punto, pero puede decirse que los restantes

códigos de condición se pueden recuperar por medio de la repetición de la instrucción `addcc` antes de la operación sobre `%r7`, tomando en cuenta el hecho de que sigue siendo necesario preservar el valor del código de condición `c`. •

```
! Realiza una suma en 64 bits: C ←A + B
! Asignación de registros:%r1 - 32 bits más significativos de A
!                 %r2 - 32 bits menos significativos de A
!                 %r3 - 32 más significativos de B
!                 %r4 - 32 bits menos significativos de B
!                 %r5 - 32 más significativos de C
!                 %r6 - 32 bits menos significativos de C
!                 %r7 - Utilizado para reponer el bit de arrastre
          .begin              ! Comienzo del ensamble
          .org  2048          ! El programa comienza en 2048
main:     ld    [A], %r1      ! Buscar la palabra más significativa de A
          ld    [A+4], %r2    ! Buscar la palabra menos significativa de A
          ld    [B], %r3      ! Buscar la palabra más significativa de B
          ld    [B+4], %r4    ! Buscar la palabra menos significativa de B
          call  add_64        ! Realizar la suma en 64 bits
          st    %r5, [C]      ! Almacenar la palabra más significativa de C
          st    %r6, [C+4]    ! Almacenar la palabra menos significativa de C
                .
                .
                .
         .org  3072           ! La rutina add_64 comienza en 3072
add_64:  addcc %r2, %r4, %r6 ! Sumar las palabras menos significativas
          bcs   lo_carry      ! Bifurcar si el arrastre vale 1
          addcc %r1, %r3, %r5 ! Sumar las palabras más significativas
          jmpl  %r15 + 4, %r0 ! Volver al programa principal
lo_carry: addcc %r1, %r3, %r5 ! Sumar las palabras más significativas
          bcs   hi_carry      ! Bifurcar si el arrastre vale 1
          addcc %r5, 1, %r5   ! Sumar el arrastre
          jmpl  %r15, 4, %r0  ! Volver al programa principal
hi_carry: addcc %r5, 1, %r5   ! Sumar el arrastre
          sethi #3FFFFF, %r7  ! Preparar %r7 para el arrastre
          addcc %r7, %r7, %r0 ! Generar arrastre
          jmpl  %r15 + 4, %r0 ! Volver al programa principal
A:                      0     ! 32 bits más significativos de 25
                       25     ! 32 bits menos significativos de 25
B:               #FFFFFFFF    ! 32 bits más significativos de -1
                 #FFFFFFFF    ! 32 bits menos significativos de -1
C:                      0     ! 32 bits más significativos del resultado
                        0     ! 32 bits menos significativos del resultado
          .end                ! Fin de la traducción
```

Figura 5.8 • Un programa ARC que suma dos enteros de 64 bits.

5.4 Macroinstrucciones (Macros)

Si las estructuras de llamado utilizan una convención basada en el uso de la pila, suele ser frecuente la necesidad de colocar y sacar de la pila una buena cantidad de registros durante los llamados a subrutina y sus retornos. Para poder almacenar en la pila de ARC el registro `%r15`, se hace necesario decrementar primero el puntero a la pila (que se encuentra en `%r14`) y luego copiar `%r15` a la posición de memoria a la que apunta `%r14`, según se muestra en el código siguiente:

```
addcc %r14, -4, %r14 ! Decrementar puntero de pila
st    %r15, %r14     ! Salvar %r15 en la pila
```

Este par de instrucciones podría expresarse en forma más compacta con una notación como la que se muestra:

```
push  %r15           ! Salvar %r15 en la pila
```

Esta forma compacta asigna nombre (`push`) a la secuencia de instrucciones que lleva a cabo la operación. El rótulo `push` suele denominarse **macro** (por macroinstrucción) en tanto que el proceso de traducir una macro a su equivalente en el lenguaje simbólico se suele conocer como **expansión de la macroinstrucción**.

```
! Definición de "push" como macroinstrucción
.macro    push arg1        ! Comienzo de la definición
addcc     %r14, -4, %r14   ! Decrementar puntero a pila
st        arg1, %r14       ! Colocar arg1 en la pila
.endmacro                  ! Fin de la definición
```

Figura 5.9 • La definición de la macroinstrucción `push`.

Una macro puede ser creada por medio de su **definición**, como se muestra en la figura 5.9 para el caso de `push`. La secuencia comienza con una directiva `.macro` y termina con otra directiva `.endmacro`. En la línea `.macro`, el primer símbolo es el nombre de la macroinstrucción a ser creada y los símbolos restantes son argumentos de la línea de comandos que se utilizan dentro de la macroinstrucción. En el caso de `push`, existe un solo argumento, al que se define como `arg1`. Corresponde a `%r15` en la sentencia "`push %r15`", o a "`%r1`" en la sentencia "`push %r1`", etc. El argumento (`%r15` o `%r1`) correspondiente a cada caso se vincula con `arg1` durante el proceso de ensamblado.

Se pueden utilizar parámetros formales adicionales, separados por comas, como sucede en

```
.macro nombre arg1, arg2, arg3, ...
```

los que, en el momento, requieren que se invoque la macro con la misma cantidad de parámetros reales*

```
nombre %r1, %r2, %r3, ...
```

A continuación de la directiva `.macro` sigue el cuerpo de la macroinstrucción. Pueden aparecer otros comandos cualesquiera, incluyendo otras macros, o incluso llamados a la misma macroinstrucción, lo que permite que se expanda en forma recursiva durante el momento del ensamblado. Los parámetros que aparecen en la línea `.macro` pueden reemplazar texto dentro del cuerpo de la macroinstrucción, por lo que pueden usarse para rótulos, instrucciones u operandos.

Debe notarse que durante la expansión de la macro los parámetros formales se reemplazan por los parámetros reales usando una simple sustitución textual. Por consiguiente, la macroinstrucción `push` puede invocarse con argumentos referidos a registros o a memoria:

```
    push    %r1
o
    push foo
```

El programador debe tener en cuenta esta característica de la expansión de macroinstrucciones al momento de definirlas, para que la macro expandida no contenga sentencias ilegales..

Si se requiere la expansión de macroinstrucciones en forma recursiva, se hacen necesarias directivas adicionales. Las directivas `.if` y `.endif`, respectivamente, abren y cierran una sección de ensamblado condicional. Si el argumento de `.if` es verdadero (al momento de la expansión de la macro), el código que aparece a continuación de `.if` se ensambla, hasta llegar a la correspondiente `.endif`. Si el argumento de `.if` es falso, el ensamblador ignora directamente el código que se encuentra entre `.if` y `.endif`. El operador de condicionalidad de la directiva `.if` puede ser cualquier elemento del grupo $\{<, =, >, \geq, \neq, o \leq\}$.

La figura 5.10 ilustra la definición recursiva de una macroinstrucción y su expansión durante el proceso de ensamblado. El código expandido suma los contenidos de los registros que van desde `%r1` hasta `%rX`, dejando el resultado de la suma en `%r1`. El argumento `X` se analiza en la línea `.if`. Si `X` es mayor que 2, la macro vuelve a ser llamada, pero con el argumento `X–1`. Si la macro `recurs_add` se invoca con un argumento igual a `4`, se generan tres líneas de código de acuerdo con lo que se muestra en la parte inferior de la figura. Cuando la macro es invocada por primera vez, `X` tiene como valor

* *N. de T.*: El hecho de que en inglés se hable de "*actual parameters*" hace que en la jerga se haya aceptado la expresión "parámetros actuales", cuando la traducción de "*actual*" corresponde al término "real".

4. La macro vuelve a ser invocada con `X = 3` y con `X = 2`, punto en el cual se genera la primera sentencia `addcc`. Las dos sentencias `addcc` siguientes se generan en la medida en que se vuelve a recurrir dentro de la macro.

Tal como se mencionara anteriormente, en el caso de un programa ensamblador que soporta macroinstrucciones, debe existir una etapa de expansión de las mismas, la que debe llevarse a cabo antes de las dos vueltas del proceso de ensamblado. La expansión de las macros se realiza normalmente por medio de un **preprocesador de macroinstrucciones** antes del ensamblado del programa. Este proceso de expansión, no obstante, puede ser invisible al programador, dado que puede ser invocado por el ensamblador. La expansión de las macroinstrucciones requiere, generalmente, dos pasos, de los cuales el primero registra las definiciones de las mismas en tanto que el segundo genera las sentencias en el lenguaje simbólico. No obstante, si se aceptan definiciones recursivas, el segundo paso de la expansión de las macroinstrucciones puede llegar a ser muy complicado. J. J. Donovan presenta una descripción más detallada del proceso de expansión de macroinstrucciones.

```
! Definición de una macro recursiva
.macro  recurs_add X          ! Comienzo de la definición
.if     X > 2                 ! Ensamblar si X > 2
        recurs_add   X - 1    ! Llamada recursiva
.endif                        ! Fin de la construcción condicional.if
        addcc  %r1, %rX, %r1  ! Sumar argumento con %r1
.endmacro                     ! Fin de la definición
-------------------------------------------------------------------
recurs_add     4              ! Invocación de la macroinstrucción
        Se expande en:
addcc   %r1, %r2, %r1
addcc   %r1, %r3, %r1
addcc   %r1, %r4, %r1
```

Figura 5.10 • Una definición recursiva de una macroinstrucción y su correspondiente expansión.

5.5 Estudio de un caso: extensiones al juego de instrucciones- Las instrucciones SIMD Intel MMX™ y Motorola AltiVec™

A medida que la tecnología de los circuitos integrados ofrece prestaciones crecientes en la estructura de los procesadores, los fabricantes buscan nuevas formas de aprovecharlas. Una de las formas mediante las cuales tanto Intel como Motorola capitalizaron dichas prestaciones adicionales fue la expansión de sus arquitecturas de programación con nuevos registros e instrucciones especializadas para el procesamiento de cadenas o bloques de datos. Intel agrega a sus procesadores Pentium la llamada extensión MMX, en tanto que Motorola agrega la extensión AltiVec en sus procesadores PowerPC. Esta sección analiza la utilidad de las extensiones y la implementación realizada por las dos compañías.

5.5.1 Fundamentos

El procesamiento de flujos de datos referidos a gráficos, audio y comunicaciones requiere la ejecución de operaciones repetitivas sobre grandes bloques de información. Por ejemplo, una imagen gráfica de varios megabytes puede requerir operaciones repetitivas sobre toda la imagen tendientes al filtrado, a la mejora de la calidad de imagen, o a algún otro tipo de proceso. El audio que se trasmite por una red en tiempo real puede requerir operaciones continuas sobre el bloque de datos a medida que se lo recibe. En forma similar, la generación de imágenes en tres dimensiones, los ambientes de realidad virtual y aun algunos juegos requieren una gran cantidad de potencia de procesamiento. En épocas anteriores, la solución adoptada por muchos fabricantes de sistemas de computación pasaba por incluir procesadores especialmente preparados para manejar este tipo de operaciones.

Si bien las soluciones adoptadas por Intel y Motorola son ligeramente diferentes en su enfoque, los resultados son relativamente similares. Los dos juegos de instrucciones se amplían con instrucciones y tipos de datos **SIMD** (*single instruction stream/multiple data stream*, único flujo de instrucciones y múltiples flujos de datos). El enfoque SIMD aplica la misma instrucción a un **vector** de elementos de datos en forma simultánea. El término vector se refiere a un conjunto de datos, habitualmente bytes o palabras.

Los procesadores vectoriales y las extensiones del procesador no son de manera alguna conceptos nuevos. Las primeras series de computadoras CRAY e IBM 370 tenían operaciones o extensiones vectoriales. De hecho, estos sistemas poseían una capacidad de procesamiento de vectores mucho más poderosa que los primeros sistemas Intel o Motorola basados en microprocesadores. No obstante, las extensiones Intel y Motorola ofrecen un considerable aumento de velocidad en las operaciones para las cuales han sido diseñadas. Estas extensiones se desarrollan con más detalle en el análisis siguiente, para el cual se presenta la figura 5.11 como una introducción al proceso. La figura muestra la instrucción PADDB (*Packed Add Bytes*) de Intel, que realiza una suma de 8 bits sobre el vector de ocho bytes contenido en el registro MM0 con el vector de ocho bytes contenido en el registro MM1, almacenando el resultado en el registro MM0.

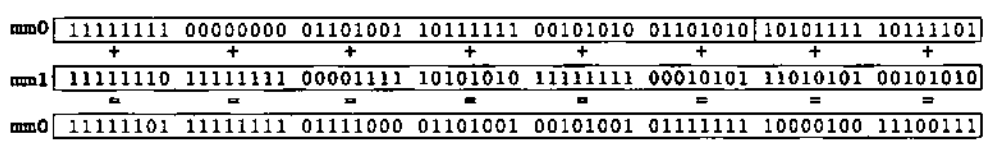


Figura 5.11 • Suma vectorial de ocho bytes a través de la instrucción Intel PADDB mm0, mm1.

5.5.2 Las arquitecturas básicas

Antes de analizar las extensiones SIMD de los dos procesadores, corresponde echar una mirada a las arquitecturas de base de las dos máquinas. Sorprendentemente, los dos procesadores no podían ser más diferentes en sus arquitecturas de programación.

El procesador Pentium Intel

Además de los registros dedicados que se utilizan en cuestiones asociadas con el sistema operativo, la arquitectura Pentium contiene ocho registros de 32 bits para manejo de enteros, cada uno de los cuales posee su propia "personalidad". Por ejemplo, la arquitectura de programación del Pentium contiene un único acumulador (EAX), utilizado para almacenar operandos y resultados aritméticos. El procesador incluye asimismo ocho registros de 80 bits para punto flotante, los que, como se verá, sirven también como registros vectoriales para las instrucciones MMX. El juego de instrucciones del Pentium debería caracterizarlo como un procesador **CISC** (computadora con un conjunto de instrucciones complicado, *complicated instruction set computer*). El análisis de las arquitecturas CISC y RISC (computadora con un conjunto de instrucciones reducido, *reduced instruction set computer*) se desarrollará con más detalle en el capítulo 10. Por ahora, es suficiente con decir que las instrucciones del procesador Pentium varían en tamaño desde un solo byte hasta 9 bytes, y que muchas instrucciones del Pentium ejecutan acciones por demás complicadas. El procesador tiene muchos modos de direccionamiento y la mayor parte de sus instrucciones aritméticas permiten que uno de los operandos o el resultado estén almacenados tanto en memoria central como en registros de la CPU. Gran parte de la arquitectura Intel está encuadrada en la decisión de permitir la compatibilidad binaria con los miembros más antiguos de la familia, como el 8086/8088, introducidos en el mercado en 1978. (La arquitectura del 8086 en sí misma también fue encuadrada en el propósito de plantear una compatibilidad a nivel de lenguaje simbólico con el venerable procesador 8080, de ocho bits, creado en 1973.)

El procesador PowerPC Motorola

El procesador PowerPC, en contraste con el anterior, fue desarrollado por un consorcio formado por IBM, Motorola y Apple, "desde los cimientos", resignando la compatibilidad con dispositivos ya existentes en aras de incorporar las novedades de la tecnología RISC. El resultado fue una arquitectura de programación con menor cantidad de instrucciones, más sencillas, y todas estructuradas en una sola palabra de 32 bits. La arquitectura incluye 32 registros de uso general, de 32 bits cada uno, para uso de enteros, y otros 32 registros, de 64 bits cada uno, para utilización en punto flotante. La arquitectura utiliza las instrucciones de cargar y almacenar (*load-store*) para acceder a la memoria. Los operandos almacenados en memoria deben cargarse en los registros por medio de instrucciones de carga y almacenamiento antes de poder ser utilizados. Todas las demás instrucciones deben acceder a sus operandos y resultados en los registros del procesador.

Como se verá, la influencia principal que las arquitecturas básicas descriptas tienen sobre las operaciones en vectores se relacionan con la forma en que acceden a la memoria.

5.5.3 Registros vectoriales

Ambas arquitecturas incluyen un conjunto adicional de registros dedicados en los que se almacenan los operandos y resultados de operaciones con vectores. La figura 5.12 muestra los conjuntos de registros vectoriales de los dos procesadores. Intel, probablemente por cuestiones de espacio, "disfraza" sus registros de punto flotante como registros MMX. Esto significa que los ocho registros de punto flotante de 64 bits del procesador Pentium cumplen doble función, no solo como registros para aplicaciones de punto flotante sino también como registros MMX. Esta solución tiene una desventaja: los registros pueden utilizarse para un único tipo de operación por vez. El conjunto de registros debe vaciarse con una instrucción especial, EMMS (*Empty MMX State*, vaciar el estado del conjunto MMX), después de ejecutar instrucciones MMX y antes de ejecutar instrucciones de punto flotante.

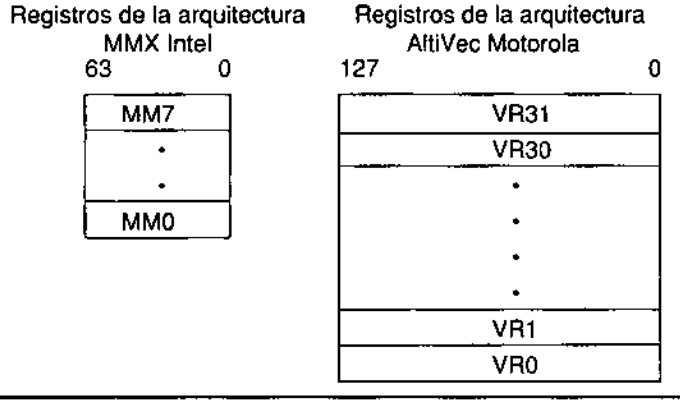


Figura 5.12 • Registros vectoriales Intel y Motorola.

Motorola, quizás debido a que su procesador PowerPC ocupa menos superficie de silicio, implementa treinta y dos registros vectoriales de 128 bits cada uno como un conjunto nuevo, separado y diferente de sus registros de punto flotante.

Operandos vectoriales

Tanto las operaciones vectoriales de Intel como las de Motorola pueden efectuarse sobre valores enteros de 8, 16, 32 y 64 bits, y, solo en el caso de Motorola, también en 128 bits. A diferencia de Intel, que no soporta más que vectores de valores enteros, Motorola también admite operaciones y números de 32 bits en punto flotante. Ambos registros vectoriales pueden llenarse o empaquetarse con datos de 8, 16, 32, 64, y, en el caso de Motorola, también de 128 bits. Para el caso de operandos de ocho bits, esta arquitectura representa un paralelismo de 8 o 16 vías, dado que el procesador opera simultáneamente sobre 8 o 16 bytes. La naturaleza SIMD de la operación vectorial puede expresarse diciendo que el procesador realiza la misma operación sobre todos los objetos de un registro vectorial dado.

Carga y descarga de los registros vectoriales

El enfoque CISC de los procesadores Intel se observa también cuando se analiza la forma de carga de sus operandos en los registros vectoriales. Existen dos instrucciones que permiten la carga y la descarga de datos hacia y desde los registros vectoriales, MOVD y MOVQ, las que mueven palabras de 32 y 64 bits, respectivamente. (Intel emplea una palabra cuyo tamaño es de 16 bits.) La sintaxis es

```
MOVD    mm, mm/m32    ;mover dos palabras a un registro vectorial.
MOVD    mm/m32, mm    ;mover dos palabras desde un registro vectorial.
MOVQ    mm, mm/m64    ;mover cuatro palabras a un registro vectorial.
MOVQ    mm/m64, mm    ;mover cuatro palabras desde un registro vectorial.
```

donde

- ***mm*** representa uno de los ocho registros vectoriales MM.
- ***mm/m32*** representa uno de los registros enteros, un registro MM o una posición de memoria.
- ***mm/m64*** representa uno de los registros MM o una dirección de memoria.

Como se verá más adelante, en las operaciones aritméticas vectoriales de Intel uno de sus operandos puede estar almacenado en memoria.

Las operaciones de carga y descarga de los procesadores Motorola se mantienen fieles a su enunciada filosofía RISC. Las operaciones de carga y descarga vectorial constituyen la única forma de acceder a un operando almacenado en memoria. No hay forma de mover un operando entre los registros vectoriales y algún otro registro. Todos los operandos deben cargarse desde memoria y descargarse hacia memoria. Los códigos de operación típicos son

```
lvebx vD, rA|0, rB ;cargar byte en el registro vectorial vD, indexado
lvehx vD, rA|0, rB ;cargar media palabra en el registro vectorial vD, indexado.
lvewx vD, rA|0, rB ;cargar palabra en el registro vectorial vD, indexado.
lvx   vD, rA|0, rB ;cargar palabra doble en el registro vectorial vD.
```

en los cuales `vD` representa uno de los 32 registros vectoriales. La dirección de memoria del operando se calcula a partir de *(*`rA | 0 + rB`*)*, en la que `rA` y `rB` representan cualquier par de registros enteros `r0-r32`, y donde el símbolo `| 0` significa que puede reemplazarse `rA` por el valor cero. Las palabras de 8 bits (byte), 16 bits (media palabra), 32 bits (palabra) y 64 bits (palabra doble) se obtienen desde su dirección de memoria. (El tamaño de las palabras del PowerPC es de 32 bits.)

El uso de la palabra "indexado" en la lista anterior se refiere a la posición dentro del registro vectorial en que se almacenarán los datos de 8, 16 y 32 bits. Los bits menos sig-

nificativos de la dirección de memoria especifican el índice a utilizar en el registro vectorial. Por ejemplo, si los tres últimos bits de la dirección son 011, estarían especificando que el byte debe almacenarse en el tercer byte del registro. Los otros bytes del registro vectorial quedan indefinidos.

Las operaciones de almacenamiento (*store*) trabajan tal como lo hacen las instrucciones de carga (*load*). La función de las mismas es almacenar en memoria el valor de uno de los registros vectoriales.

5.5.4 Operaciones de aritmética vectorial

Las operaciones de aritmética vectorial forman el corazón del proceso SIMD. Del análisis de las mismas surgirán, por un lado, una nueva forma de aritmética, la **aritmética de saturación**, y por el otro, algunas operaciones exóticas y nuevas.

Aritmética de saturación

Ambos procesadores vectoriales ofrecen la opción de utilizar la llamada **aritmética de saturación** en lugar de la aritmética más familiar que opera en formato al módulo, y que fuera analizada en los capítulos 2 y 3. La aritmética de saturación funciona en la misma forma que la aritmética de complemento a dos en tanto no se produzca desborde en los resultados. Cuando los resultados obtenidos superan los límites permitidos, tanto máximo como mínimo, el resultado se mantiene en el valor máximo o mínimo permitido. Por ejemplo, una representación de ocho bits en complemento a dos satura en +127 en su extremo superior y en -128 en su límite inferior. Las representaciones de ocho bits sin signo saturan en +255 y en 0. Si el resultado de una operación aritmética desborda esos límites, entonces este se recorta o "satura" en la frontera.

La necesidad de utilizar este tipo de aritmética surge en el procesamiento de la información de color. Si el color se representa por un byte, en el que 0 representa el negro y 255 representa el blanco, la saturación permite que el color se mantenga como blanco puro o negro puro luego de una operación, en lugar de invertirse si se diera el caso de un desborde en uno u otro sentido.

Formatos de instrucción

Así como las dos arquitecturas tienen distintos enfoques en lo que hace a sus modos de direccionamiento, también difieren en sus formatos de instrucciones SIMD. Intel sigue utilizando instrucciones de dos direcciones, donde el primer operando, origen, puede ser un registro MM, un registro entero o memoria, en tanto que el segundo operando, destino, es un registro MM.

```
OP mm,  mm32or64          ;mm ← mm OP mm/mm32/64
```

Motorola requiere que todos sus operandos se encuentren en los registros vectoriales y utiliza instrucciones de tres operandos:

```
OP Vd, Va, Vb  [,Vc]      ; Vd ← Va OP Vb [OP Vc]
```

Este enfoque tiene la ventaja de que no se requiere sobreescribir ningún registro vectorial. Además, algunas instrucciones pueden utilizar un tercer operando `Vc`.

Operaciones aritméticas

Quizás no sorprenda demasiado que las instrucciones MMX y AltiVec sean bastante similares. Ambas arquitecturas permiten operaciones sobre operandos de 8, 16, 32 y 64 bits, y, en el caso de AltiVec, también de 128 bits. La tabla 5.1 muestra ejemplos de la diversidad de operaciones que ofrecen ambas tecnologías. Algunas de las razones principales para que esto ocurra son: la necesidad de ofrecer a los potenciales usuarios aquellas operaciones que pudiesen considerar necesarias y útiles en sus aplicaciones particulares, la cantidad de silicio disponible para la ampliación y la arquitectura básica de programación.

Operación	Operandos (bits)	Aritmética
Suma y resta de enteros, con y sin signo (B)	8, 16, 32, 64, 128	Módulo, saturada
Suma y resta de enteros, almacenar arrastre de salida en registro vectorial (M)	32	Módulo
Multiplicación entera, almacenar mitad más o menos significativa (I)	16←16x16	–
Multiplicación entera: Vd = Va *Vb + Vc (B)	16←8x8 32←16x16	Módulo, saturada
Desplazamiento a izquierda, a derecha, a derecha con signo (B)	8, 16, 32, 64 (I)	–
Rotación a izquierda, derecha (M)	8, 16, 32	–
Y, NO Y, O, NOR, XOR(B)	64(I), 128(M)	–
Multiplicación de enteros sobre cualquier otro operando, almacenar resultado completo, con y sin signo (M)	16←8x8 32←16x16	Módulo, saturada
Máximo, mínimo. Vd←Max,Min(Va, Vb) (M)	8, 16, 32	Con y sin signo
Suma vectorial sobre palabras. Sumar los elementos de vector, sumar esta suma al objeto del segundo vector, colocar el resultado en el tercer registro vectorial (M)	Variados	Módulo, saturada
Operaciones vectoriales de punto flotante, suma, resta, producto-suma, etc. (M)	32	IEEE punto flotante

Tabla 5.1 • Instrucciones aritméticas MMX y AltiVec.

5.5.5 Operaciones de comparación de vectores

El paradigma habitual de las operaciones condicionales –comparar y bifurcar según una condición determinada– puede no funcionar en operaciones vectoriales, dado que cada operando sobre el que se ejecute la comparación puede dar resultados diferentes. Por ejemplo, la comparación por igualdad de dos vectores formados por palabras puede dar resultados como CIERTO, FALSO, FALSO, CIERTO. No hay forma adecuada de emplear instrucciones de bifurcación para seleccionar diferentes bloques de código en función de la verdad o falsedad de las comparaciones. Como consecuencia, las comparaciones vectoriales tanto en las tecnologías MMX como AltiVec dan por resultado la generación explícita de CIERTO o FALSO. En ambos casos, CIERTO se expresa con una palabra formada totalmente por unos y FALSO se expresa con una palabra formada por ceros, almacenada en el operando de destino. Por ejemplo, las comparaciones entre bytes dan por resultado FFH o 00H, los cálculos de 16 bits dan FFFFH o 0000H, y resultados similares para otros operandos. Estos valores formados por todos unos o todos ceros pueden utilizarse luego como máscaras para actualizar valores.

Ejemplo: comparar por igualdad dos vectores formados por bytes

Considérese la comparación de dos vectores MMX formados por bytes con el objeto de determinar la igualdad de los mismos. La figura 5.13 muestra el resultado de la comparación: cadenas de unos cuando la comparación es exitosa y ceros donde falla. Esta comparación puede utilizarse en operaciones subsiguientes. Considérese la sentencia condicional de un lenguaje de alto nivel:

```
if (mm0 == mm1) mm2 = mm2 else mm2 = 0;
```

La comparación de la figura 5.13 produce la máscara que puede utilizarse para controlar la asignación de bytes. Se realiza una operación lógica AND entre el registro mm2 y la máscara contenida en mm0, y se almacena el resultado en mm2, tal como se muestra en la figura 5.14. Puede implementarse una gran variedad de operaciones condicionales por medio de la utilización de distintas combinaciones de operaciones de comparación y de máscaras.

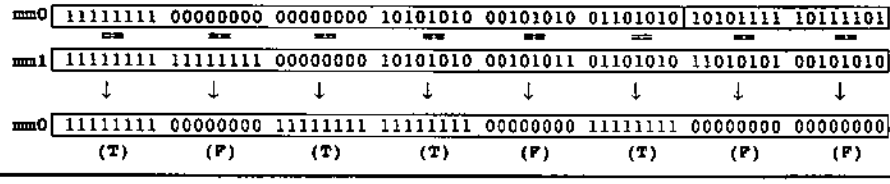


Figura 5.13 • Comparación por igual de dos vectores MMX formados por bytes.

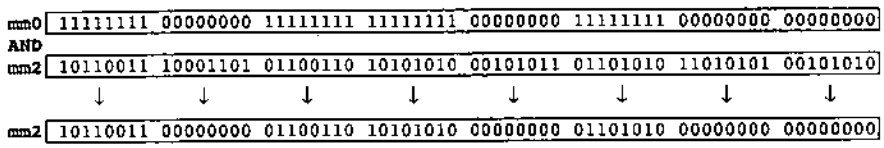


Figura 5.14 • Asignación condicional de un vector MMX formado por bytes.

Operaciones de permutación de vectores

La arquitectura de programación AltiVec incluye también una instrucción muy útil que permite permutar, o reordenar, los contenidos de un vector en forma arbitraria, almacenando el resultado en otro registro vectorial.

5.5.6 Conclusiones de los casos de estudio

Las extensiones SIMD de los procesadores Pentium y PowerPC ofrecen operaciones poderosas que pueden utilizarse para el procesamiento de bloques de datos. A la fecha de edición de la presente* no existen extensiones de compilación para estas instrucciones. Como consecuencia, aquellos programadores que quieran usar estas extensiones tienen que estar dispuestos a programar en lenguaje simbólico.

Surge un problema adicional debido a que no todas las versiones de los procesadores Pentium o PowerPC contienen estas extensiones, las que solo aparecen en algunas versiones especializadas. Dado que el programador puede verificar la presencia de las extensiones, en su ausencia el programador debe escribir una versión "manual" del algoritmo. Esto implica generar dos conjuntos de instrucciones, uno que utiliza las extensiones y otro que utiliza la arquitectura básica de programación.

Resumen

Los lenguajes de programación como C o Pascal permiten que la arquitectura de bajo nivel de la computadora sea tratada como un ente abstracto. Los lenguajes simbólicos, por otra parte, adoptan un aspecto fuertemente dependiente de la arquitectura subyacente. La arquitectura de programación se hace visible al programador, quien es responsable por el uso de registros y por las llamadas a subrutinas. Parte de la complejidad de la programación en lenguaje de máquina se administra a través de macroinstrucciones, las cuales difieren de las subrutinas o funciones en que las macroinstrucciones generan código al momento del ensamblado, mientras que las subrutinas se resuelven en el momento de la ejecución.

* *N. de T.*:Los autores se refieren a la edición original en inglés, año 2000.

Un programa de enlace combina módulos ensamblados por separado en un único módulo de carga, lo que normalmente implica código reubicable. Un programa cargador coloca el módulo de carga en memoria y comienza la ejecución del programa. El cargador puede necesitar reubicar código si detecta la superposición en memoria de dos o mas módulos.

En la práctica, los detalles de ensamblado, enlace y carga son fuertemente dependientes del sistema y del lenguaje. Algunos ensambladores simples tan solo producen programas binarios ejecutables, pero en la mayor parte de los casos, el ensamblador generará información adicional para permitir el enlace de los módulos por parte de un *linker*. Algunos sistemas proveen cargadores enlazadores que combinan las tareas de enlace y de carga. Otros separan el enlace de la carga. Algunos cargadores solo pueden cargar un programa en la dirección especificada en el archivo binario, mientras que es más habitual que los cargadores reubicadores puedan reubicar los programas en una dirección especificada en el momento de la carga. Los formatos de archivo que soportan estos procesos también dependen del sistema operativo.

Antes del desarrollo de los compiladores, los programas se escribían directamente en lenguaje simbólico. Hoy en día, el lenguaje simbólico no es de uso habitual debido a que los compiladores para lenguajes de alto nivel generan código eficiente, aun cuando el lenguaje de bajo nivel sigue siendo importante para entender algunos aspectos de la arquitectura de la computadora como, por ejemplo, la forma de vincular programas que fueron compilados para distintas convenciones de llamada, y para aprovechar las extensiones de arquitecturas de programación como, por ejemplo, las MMX y AltiVec.

Para lectura posterior

Los textos de A. V. Aho y otros, y de W. M. Waite y L. R. Carter tratan el tema de los compiladores y la compilación. Existen muchísimas referencias sobre el tema de la programación en lenguaje simbólico. J. J. Donovan es un clásico en lo que hace a ensambladores, *linkers* y cargadores. A. Gill y otros analiza el procesador 68000. El de J. Goodman y K. Miller es un buen texto de estudio, con ejemplos tomados de la arquitectura MIPS. El apéndice de D. A. Patterson y J. L. Hennessy también analiza la arquitectura MIPS. El texto de SPARC International Inc. trata específicamente la definición del procesador SPARC y su lenguaje de máquina.

Aho, A. V., R. Sethi y J. D. Ullman, *Compilers*, Addison-Wesley, 1985.

Donovan, J. J., *Systems Programming*, McGraw-Hill, 1972.

Gill, A., E. Corwin y A. Logar, *Assembly Language Programming for the 68000*, Prentice Hall, 1987.

Goodman, J. y K. Miller, *A Programmer's View of Computer Architecture*, Saunders College Publishing, 1993.

Patterson, D. A. y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 2ª ed., Morgan Kaufmann, 1998.

SPARC International Inc., *The SPARC Architecture Manual: Versión 8*, Prentice Hall, 1992.

Waite, W. M. y L. R. Carter, *An Introduction to Compiler Construction*, Harper Collins, 1993.

Problemas

5.1 Crear una tabla de símbolos para el segmento de programa ARC que se muestra a continuación, usando un formato similar al de la figura 5.7. Utilizar "U" para los símbolos indefinidos.

```
x          .equ     4000
           .org     2048
           ba       main
           .org     2072
main:      sethi    x, %r2
           srl      %r2, 10, %r2
lab_4:     st       %r2, [k]
           addcc    %r1, -1, %r1
foo:       st       %r1, [k]
           andcc    %r1, %r1, %r0
           beq      lab_5
           jmpl     %r15 + 4, %r0
cons:      .dwb     3
```

5.2 Traducir a código objeto el código ARC del programa siguiente. Asumir la dirección $(4096)_{10}$ como valor inicial de x.

```
k          .equ     1024
               .
               .
               .
           addcc    %r4 + k, %r4
           ld       %r14, %r5
           addcc    %r14, -1, %r14
           st       %r5, [x]
               .
               .
               .
```

5.3 Crear una tabla de símbolos para el programa de la figura 5.8, siguiendo un criterio similar al de la figura 5.7

5.4 Traducir a código objeto la subrutina `add_64`, de la figura 5.8, incluyendo las variables A, B y C.

5.5 Un **desensamblador** es un programa que lee un módulo objeto y recrea el módulo fuente en lenguaje simbólico. Dado el siguiente código objeto, desensamblarlo para obtener las sentencias correspondientes del lenguaje simbólico ARC. Dado que el código objeto no contiene información suficiente para determinar los nombres de los símbolos, se les asignarán ordenadamente las letras del abecedario a medida que sean necesarias.

```
1000 0010 1000 0000 0110 0000 0000 0001
1000 0000 1001 0001 0100 0000 0000 0110
0000 0010 1000 0000 0000 0000 0000 0011
1000 1101 0011 0001 1010 0000 0000 1010
0001 0000 1011 1111 1111 1111 1111 1100
1000 0001 1100 0011 1110 0000 0000 0010
```

5.6 Dadas las siguientes macroinstrucciones `push` y `pop`, si en un programa se utiliza `push` inmediatamente después de `pop` pueden aparecer instrucciones innecesarias. Expandir las definiciones de las macroinstrucciones mostradas e identificar las instrucciones innecesarias.

```
        .begin
        .macro    push    arg1
        addcc     %r14, -4, %r14
        st      arg1,  %r14
        .endmacro
        .macro    pop     arg1
        ld        %r14, arg1
        addcc     %r14, 4, %r14
        .endmacro
    !Comienzo de programa
        .org      2048
        pop       %r1
        push      %r2
            .
            .
            .
        .end
```

5.7 Escribir una macroinstrucción `return` que cumpla la función de la sentencia `jmpl` tal como se la usa en la figura 5.5.

5.8 En la figura 4.16, el operando `x` de `sethi` se completa desde el ensamblador. La sentencia no funciona como se espera si $x \geq 2^{22}$ debido a que solo hay 22 bits en el campo `imm22` del formato `sethi`. Para poder colocar una dirección arbitraria de 32 bits en `%r5` al momento de la ejecución, puede utilizarse `sethi` para los 22 bits superiores, usando luego `addcc` para los 10 bits inferiores. Esto requiere agregar dos nuevas directivas, `.high22` y `.low10`, las que construyen las palabras binarias para los 22 bits más significativos y para los 10 bits menos significativos de la dirección, respectivamente. La construcción

```
sethi .high22 (#FFFF FFFF), %r1
```

se expande a

```
sethi #3FFFFF, %r1
```

en tanto que la estructura

```
addcc %r1, .low10(#FFFF FFFF), %r1
```

se expande a

```
addcc %r1, #3FF, %r1
```

Reescribir la rutina de llamado de la figura 4.16 utilizando `.high22` y `.low10` de modo que funcione adecuadamente sin importar la posición de `x` en memoria.

5.9 Supóngase tener disponible para su uso la rutina `add_64` de la figura 5.8. Se requiere escribir una rutina ARC llamada `add_128` que sume dos números de 128 bits, haciendo uso de la rutina `add_64`. Los dos operandos de 128 bits se encuentran almacenados en memoria, en direcciones que empiezan en `x` e `y`, respectivamente, en tanto que el resultado se almacena en la dirección de memoria que comienza en `z`.

5.10 Escribir una macroinstrucción llamada `subcc` cuya utilidad sea similar a la de `addcc`, que realice la resta del primer operando origen menos el segundo.

5.11 ¿Cuándo se produce la expansión de las macroinstrucciones no recursivas, en el momento del ensamblado o en el momento de la ejecución? ¿Cuándo se produce la expansión de las macroinstrucciones recursivas?

5.12 Un programador de lenguaje simbólico propone incrementar la prestación de la macro `push`, definida en la figura 5.9, por medio del agregado de un segundo argumento, `arg2`. El segundo argumento reemplazaría a la instrucción `addcc %r14, -4, %r14` con `addcc arg2, -4, arg2`. Explicar qué es lo que está intentando el programador y qué peligros se esconden en este enfoque.

Capítulo 6

Trayecto de datos y control

Los capítulos anteriores permitieron examinar la computadora desde el nivel de las aplicaciones, desde el nivel de los lenguajes de alto nivel y desde el nivel del lenguaje de máquina (según la clasificación de la figura 1.4). En el capítulo 4 se introdujo el concepto de arquitectura de programación: un conjunto de instrucciones que realiza operaciones sobre los registros y sobre la memoria. En este capítulo se analiza la parte de la máquina que es responsable de la implementación de estas operaciones: la unidad de control de la unidad central de proceso. En este contexto, se analiza la máquina en el marco de su **microarquitectura** (el nivel correspondiente al control microprogramado o cableado de la figura 1.4). La microarquitectura está constituida por la unidad de control y los registros accesibles al programador, las unidades funcionales del tipo de la unidad aritmético-lógica y todo otro registro adicional que la unidad de control pueda requerir.

Una arquitectura de programación determinada puede implementarse con microarquitecturas diferentes. Por ejemplo, la arquitectura de programación Pentium de Intel se ha implementado en distintas formas, aun cuando todas soportan la misma arquitectura de programación. No solo Intel ha implementado una arquitectura de programación Pentium, sino que lo han hecho también varios competidores como AMD y Cyrix. Algunas de las microarquitecturas pueden apuntar a mejorar la velocidad de ejecución del juego de instrucciones, otras pueden apuntar a disminuir el consumo de energía y otras más pueden tener como objetivo disminuir el costo del procesador. La posibilidad de modificar la microarquitectura sin modificar la arquitectura de programación implica que los fabricantes pueden aprovechar las nuevas tecnologías de fabricación de memorias y de circuitos integrados en general, para ofrecer a sus usuarios compatibilidad en los desarrollos de software. Un programa se ejecuta sin modificación sobre diferentes procesadores siempre que los procesadores implementen la misma arquitectura de programación, independientemente de la microarquitectura subyacente.

Este capítulo tiene como objetivo examinar dos enfoques de microarquitectura totalmente diferentes: el de las unidades de control microprogramadas y el de las unidades de control cableadas, para lo cual se mostrará cómo puede implementarse un subconjunto del procesador ARC utilizando estas dos técnicas de diseño.

6.1 Fundamentos de la microarquitectura

La funcionalidad de una microarquitectura se centra en el ciclo de búsqueda y ejecución de una instrucción, el que, de alguna manera, puede pensarse como el corazón de la máquina. Tal como se analizara en el capítulo 4, los pasos involucrados en el ciclo búsqueda-ejecución son los siguientes:

1. Buscar en memoria la siguiente instrucción que debe ejecutarse.
2. Decodificar el código de operación.
3. Si los hubiera, leer los operandos desde memoria o desde registros.
4. Ejecutar la instrucción y almacenar los resultados.
5. Volver al paso 1.

La ejecución de estos cinco pasos es responsabilidad de la microarquitectura. Es la microarquitectura la que busca a la instrucción que debe ser ejecutada, determina de qué instrucción se trata, localiza los operandos, ejecuta la instrucción, almacena los resultados y, finalmente, repite la secuencia para la instrucción siguiente.

La microarquitectura de una máquina incluye una **sección de datos**, que contiene registros y una unidad aritmético-lógica, y una **sección de control**, como se ilustra en la figura 6.1. La sección de datos se conoce habitualmente como **trayecto de datos**. El control microprogramado utiliza un **microprograma** especialmente dedicado a estos efectos, no visible al usuario, el que implementa las operaciones sobre los registros y sobre otros sectores de la máquina. A menudo, el microprograma contiene muchos pasos de programa que en su conjunto implementan una única (macro)instrucción. Las unidades de control **cableadas** adoptan como enfoque el de generar los distintos pasos requeridos para implementar una operación como estados sucesivos de una máquina de estados finitos, por lo que el diseño de las mismas involucra técnicas de diseño digital convencionales (como las que se plantean en el apéndice A). En cualquiera de los dos casos, los trayectos de datos se mantienen casi sin modificación alguna, aun cuando puede haber diferencias mínimas para admitir las

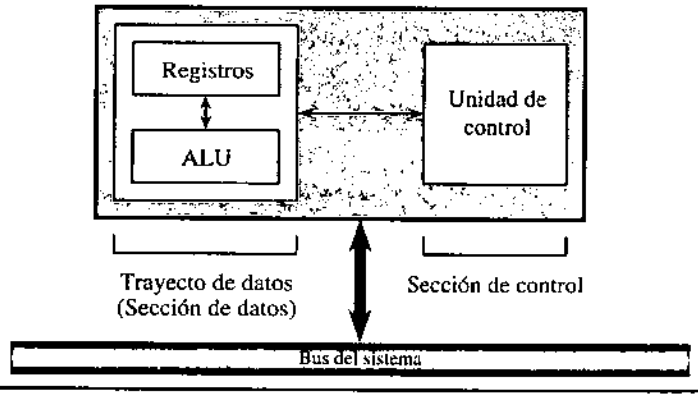


Figura 6.1 • Microarquitectura vista desde el alto nivel.

distintas formas de control. Al diseñar la unidad de control de ARC, primero se analizará el enfoque microprogramado y luego se desarrollará una estructura cableada. En ambos casos, el trayecto de datos será el mismo.

6.2 Una microarquitectura para ARC

En esta sección se considera el enfoque microprogramado para diseñar la unidad de control de ARC. El enfoque comienza describiendo el trayecto de datos y sus señales de control asociadas.

En la figura 6.2 se repiten el conjunto y el formato de instrucciones de ARC descriptos en el capítulo 4. El juego de instrucciones incluye 15 instrucciones agrupadas en cuatro formatos definidos por los dos bits más significativos de la instrucción codificada. Asimismo, se muestra el formato del registro de estado del procesador (`%psr`).

6.2.1 El trayecto de datos

La figura 6.3 ilustra un trayecto de datos en ARC. El mismo contiene 32 registros de datos accesibles por el usuario (`%r0-%r31`), el contador de programa (`%pc`), el registro de instrucciones (`%ir`), la unidad aritmético-lógica (ALU), cuatro registros temporarios no accesibles al nivel de la arquitectura de programación (`%temp0-%temp3`) y las conexiones entre estos elementos. En la figura se simplifica la representación de los conjuntos de líneas bajo la forma de un número acompañado por una barra diagonal sobre la única línea que representa al conjunto completo.

Los registros `%r0-%r31` son accesibles directamente por el usuario. El registro `%r0` siempre contiene un 0, y esto no puede modificarse. El `%pc` es el contador de programa, que apunta a la dirección a ser leída desde la memoria principal. El usuario tiene acceso directo al contador de programa solo a través de las instrucciones `call` y `jmpl`. Los registros temporarios se utilizan para interpretar el conjunto de instrucciones de ARC, por lo que no son accesibles para el usuario. El registro `%ir` contiene la instrucción en ejecución, y tampoco es accesible al usuario.

La unidad aritmético-lógica

La unidad aritmético-lógica puede realizar una de 16 operaciones sobre los buses A y B, de acuerdo con la tabla de la figura 6.4. Para cada operación realizada en la unidad aritmético-lógica, el resultado de 32 bits se coloca en el bus C, a menos que quede bloqueada por el multiplexor que maneja el bus C como consecuencia de colocar en él una palabra proveniente de la memoria de la máquina.

Las instrucciones ANDCC y AND, ejecutan un producto lógico, bit a bit, de los bits homólogos en los operandos presentes sobre los buses A y B. Debe tenerse en cuenta que

Mnemónico	Significado
ld	Cargar un registro desde memoria
st	Salvar un registro en memoria
sethi	Cargar los 22 bits más significativos de un registro
andcc	Producto lógico bit por bit
orcc	Suma lógica bit por bit
orncc	Operación lógica NOR bit por bit
srl	Desplazamiento a derecha (sin signo)
addcc	Suma
call	Llamado a subrutina
jmpl	Salto y enlace (retorno desde subrutina)
be	Salto por igual
bneg	Salto por negativo
bcs	Salto si hay arrastre
bvs	Salto si hay desborde
ba	Salto incondicional

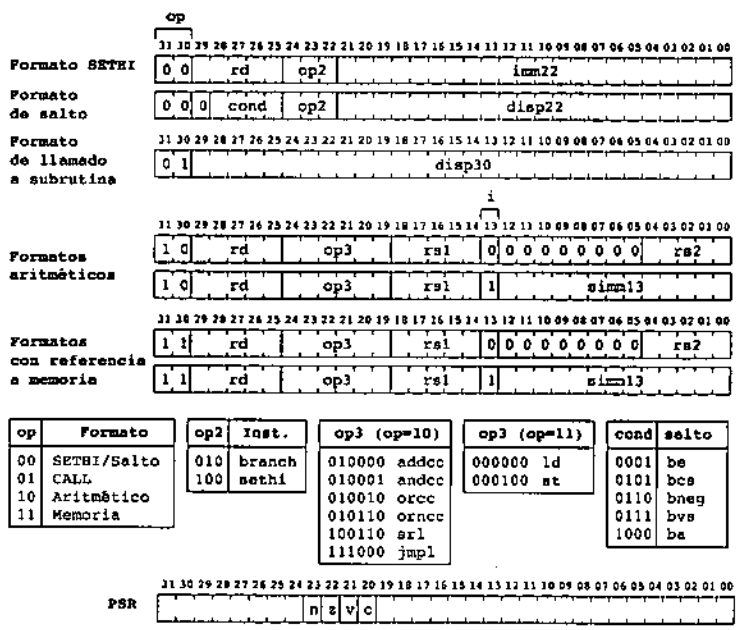


Figura 6.2 • Conjunto de instrucciones y su formato en ARC.

solo aquellas instrucciones cuyo código simbólico termina en CC afectan a los códigos de condición, por lo que ANDCC afecta el contenido del registro de estado, mientras que AND no lo hace. (Pueden producirse situaciones en las que se desee realizar operaciones aritméticas y lógicas sin afectar a los códigos de condición). Las instrucciones ORCC y OR realizan la suma lógica bit a bit de los bits correspondientes a los operandos presentes en los buses A y B. Las instrucciones NORCC y NOR realizan la operación lógica nor de los bits homólogos de los operandos presentes sobre los buses A y B. Las instrucciones ADDCC y ADD llevan a cabo la operación de suma aritmética de los operandos presentes en los buses A y B, utilizando la representación de complemento y signo de los mismos.

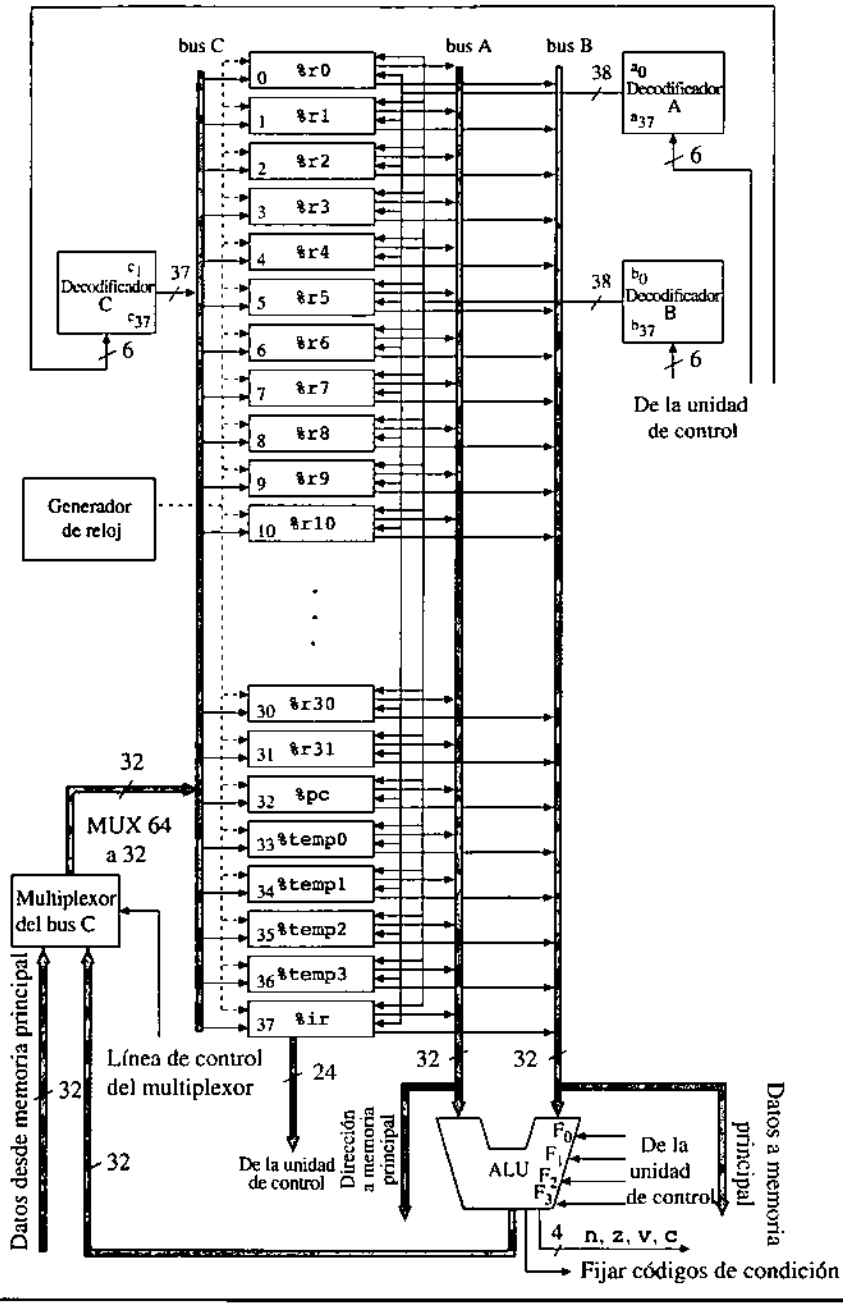


Figura 6.3 • Trayecto de datos en ARC.

La instrucción `srl` (desplazamiento lógico a derecha) desplaza el contenido del bus A hacia la derecha en una cantidad de bits especificado por el dato presente en el bus B (desde 0 hasta 31 desplazamientos). Se introducen ceros en los bits más significativos del resultado desplazado, en tanto que los bits menos significativos se descartan. `LSHIFT2`

y LSHIFT10 desplazan a izquierda el contenido del bus A, en dos y diez posiciones, respectivamente, completando con ceros los bits menos significativos.

SIMM13 recupera los 13 bits menos significativos de la palabra contenida en el bus A y coloca ceros en los 19 bits restantes. SEXT13 realiza la extensión de signo de los 13 bits menos significativos de la palabra contenida sobre el bus A para crear una palabra de 32 bits. Esto significa que si el bit más significativo del grupo de 13 bits es 1, se copiarán unos en los 19 bits restantes del resultado. En caso contrario, los 19 bits se completarán con ceros. La instrucción INC incrementa el valor contenido en el bus A en 1, en tanto que la instrucción INCPC incrementa el valor del bus A en cuatro, lo que se utiliza para aumentar el contenido del contador de programa en una palabra (cuatro bytes). INCPC puede utilizarse sobre el contenido de cualquier registro conectado con el bus A.

La instrucción RSHIFT5 desplaza en cinco lugares hacia la derecha al operando que se encuentra en el bus A, copiando en los cinco bits libres de la izquierda el bit más significativo original (el bit de signo del operando inicial). El efecto producido es el de una extensión de signo sobre los cinco bits más significativos. Al aplicarse tres veces consecutivas sobre una instrucción de 32 bits, la operación, además, coloca el bit más significativo del campo COND del formato de saltos (véase la figura 6.2) en la posición del bit 13. Esta operación es útil para la decodificación de instrucciones de salto, de acuerdo con lo que se analizará en este mismo capítulo, más adelante. En este caso, la extensión de signo producida no tiene significado o consecuencia alguna.

Cada una de las operaciones aritméticas o lógicas puede implementarse exclusivamente con estas operaciones de la unidad aritmético-lógica. Por ejemplo, una operación de resta se resuelve obteniendo el complemento a dos del sustraendo (usando la instrucción NOR y sumando 1 al resultado por medio de la instrucción INC), tras lo cual se suman los dos operandos. Se puede desplazar un número a izquierda sumándolo consigo mismo. Una operación de "no hacer nada", requerida frecuentemente solo para pasar datos a través de la unidad aritmético-lógica sin modificarlos, puede llevarse a cabo a través del producto lógico de un operando consigo mismo y el descarte del resultado obtenido a través del uso del registro %r0. Una operación lógica de XOR puede efectuarse mediante las instrucciones Y, O y NOR, haciendo uso de los teoremas de DeMorgan (véase el problema 6.5).

La unidad aritmético-lógica genera los códigos de condición c, n, z y v, que resultan ciertos cuando se produce, respectivamente, un arrastre, un resultado negativo, un resultado nulo y un desborde. Estos códigos de condición solo se modifican a través de las instrucciones que así lo indican en la tabla de la figura 6.4. En estos casos se genera una señal (SCC) que le indica al registro %psr que debe actualizar sus códigos de condición.

La unidad aritmético-lógica puede implementarse en distintas formas. Para simplificar el análisis se considera, inicialmente, un enfoque basado en una **tabla de búsqueda** (LUT, *lookup table*). La unidad aritmético-lógica tiene dos entradas de datos *A* y *B* de 32 bits ca-

F_3	F_2	F_1	F_0	Operación	Modifica códigos de condición
0	0	0	0	`ANDCC (A, B)`	sí
0	0	0	1	`ORCC (A, B)`	sí
0	0	1	0	`NORCC (A, B)`	sí
0	0	1	1	`ADDCC (A, B)`	sí
0	1	0	0	`SRL (A, B)`	no
0	1	0	1	`AND (A, B)`	no
0	1	1	0	`OR (A, B)`	no
0	1	1	1	`NOR (A, B)`	no
1	0	0	0	`ADD (A, B)`	no
1	0	0	1	`LSHIFT2 (A)`	no
1	0	1	0	`LSHIFT10 (A)`	no
1	0	1	1	`SIMM13 (A)`	no
1	1	0	0	`SEXT13 (A)`	no
1	1	0	1	`INC (A)`	no
1	1	1	0	`INCPC (A)`	no
1	1	1	1	`RSHIFT5 (A)`	no

Figura 6.4 • Operaciones aritméticas en ARC.

da una, una salida de datos C de 32 bits, una entrada F de control de cuatro bits, una salida de códigos de condición de cuatro bits (N, V, C, Z) y una señal (SCC) que actualiza los bits del registro `%psr`. La unidad aritmético-lógica puede descomponerse en una cascada de 32 tablas de búsqueda que implementan cada una de las operaciones aritméticas y lógicas, seguida de un circuito **controlador de desplazamientos** (*barrel shifter*) que implementa los desplazamientos. El diagrama en bloques correspondiente se muestra en la figura 6.5.

El circuito controlador de desplazamientos provoca el desplazamiento de la palabra de entrada en una cantidad de bits arbitraria (entre 0 y 31 posiciones), de acuerdo con lo que determinen las entradas de control. Los desplazamientos se producen por niveles, y en cada nivel se observa un bit diferente de la entrada que indica la cantidad de desplazamientos (*Shift Amount*, *SA*). La figura 6.6 presenta una configuración parcial, a nivel de la estructura lógica de compuertas, del controlador de desplazamientos. A partir de la parte inferior del circuito se puede ver que las salidas de la etapa inferior coinciden con las entradas a dicha etapa si el bit SA_0 vale 0. Si el bit SA_0 vale 1, cada posición de la salida tomará el valor de su vecino inmediato a derecha o a izquierda, dependiendo de la dirección del desplazamiento, la que a su vez se indica en la entrada de desplazamiento a derecha (*Shift Right*). En el nivel superior inmediato, se vuelve a aplicar el mismo método, pero con dos diferencias: se debe considerar el bit SA_1 y se tiene que duplicar la cantidad de desplazamientos.

El proceso se repite hasta observar el bit SA_4 en el nivel superior. En las posiciones que no tienen entradas asignadas se colocan ceros. Con esta estructura se puede implementar cualquier desplazamiento entre 0 y 31 bits, a derecha o a izquierda.

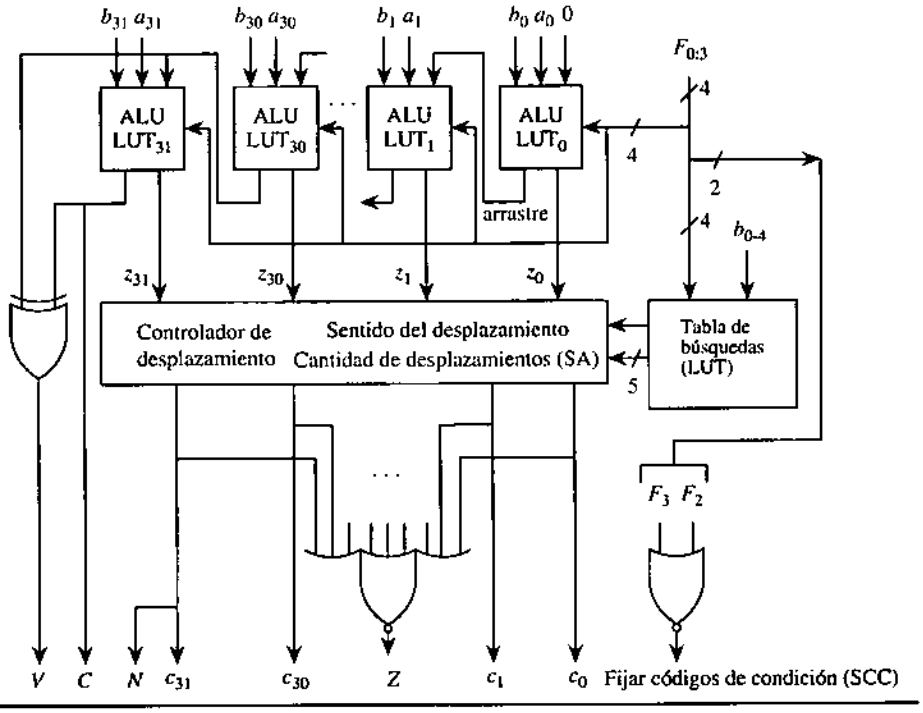


Figura 6.5 • Diagrama en bloques de la unidad aritmético-lógica de 32 bits.

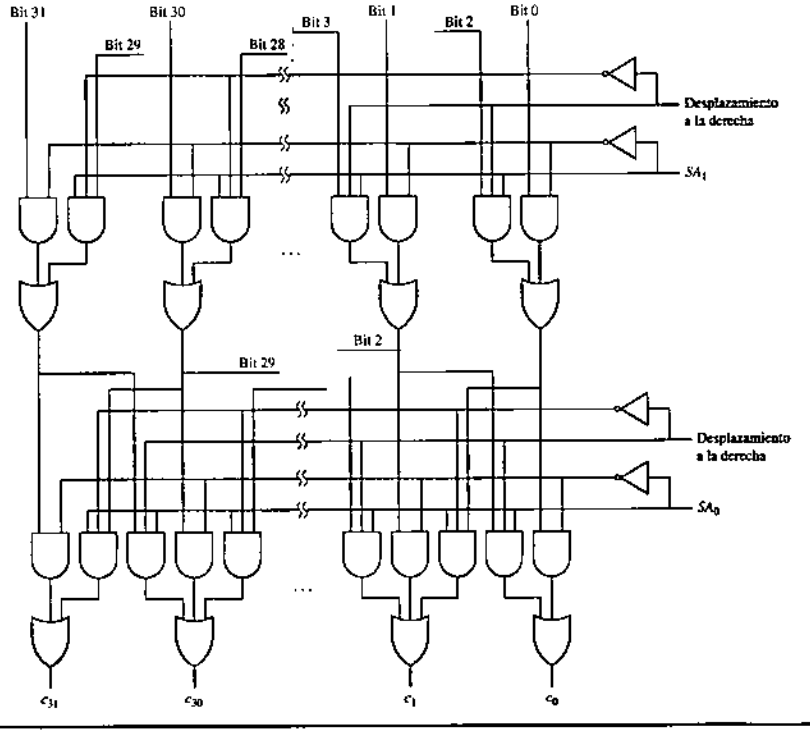


Figura 6.6 • Esquema circuital del control de desplazamiento.

Cada una de las 32 tablas de búsqueda de la unidad aritmético-lógica se implementa casi en forma idéntica, usando las mismas entradas a la tabla de búsqueda, excepto por algunos cambios requeridos en ciertas posiciones para las operaciones, por ejemplo, de INC e INCPC (véase la figura 6.7). Las primeras entradas de cada tabla de búsqueda se ilustran en la figura 6.7. La tabla de búsqueda que controla el circuito de desplazamientos se construye en forma similar, si bien las entradas a su tabla de búsqueda son diferentes.

	F_3	F_2	F_1	F_0	*Arrastre de entrada*	a_i	b_i	z_i	*Arrastre de salida*
ANDCC	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	1	1	1	0
	0	0	0	0	1	0	0	0	0
	0	0	0	0	1	0	1	0	0
	0	0	0	0	1	1	0	0	0
	0	0	0	0	1	1	1	1	0
ORCC	0	0	0	1	0	0	0	0	0
	0	0	0	1	0	0	1	1	0
	0	0	0	1	0	1	0	1	0
	0	0	0	1	0	1	1	1	0
	0	0	0	1	1	0	0	0	0
	0	0	0	1	1	0	1	1	0
					.				.
					.				.
					.				.

Figura 6.7 • Tabla de verdad para la mayoría de las tablas de búsqueda.

Los códigos de condición n, z, v y c se implementan en forma directa. Los bits n y c se obtienen directamente de la salida c_{31} del circuito controlador de desplazamiento y de la posición de arrastre que sale de la tabla de búsqueda LUT_{31} de la unidad aritmético-lógica, respectivamente. El bit z se determina a partir de la operación NOR ejecutada sobre las salidas del circuito de desplazamiento. El bit z vale 1 solo si todas las salidas del mismo son cero simultáneamente. El bit de desborde (v, *overflow*) adopta el valor 1 si el arrastre ingresado a la posición más significativa de la palabra difiere del arrastre generado desde dicha posición, lo que se implementa con una compuerta XOR.

Solo aquellas operaciones que terminen en "CC" deberían modificar los códigos de condición, por lo que debe existir una señal que indique esa situación, la que se identifica como SCC (*set condition codes*, modificar códigos de condición). Esta señal es cierta cuando tanto F_3 como F_2 son falsas.

Los registros

Todos los registros están implementados con circuitos biestables D **activados por flanco negativo** (véase el apéndice A). Esto significa que las salidas no cambiarán hasta que la señal

de sincronismo no sufra una transición desde su nivel alto a su nivel bajo (el flanco de caída del reloj). Todos los registros adoptan un formato similar, por lo que solo se analizará el diseño del registro `%r1`. Todos los registros del trayecto de datos son de 32 bits de ancho y, por consiguiente, se requieren 32 bits para el diseño de `%r1`, el que se ilustra en la figura 6.8.

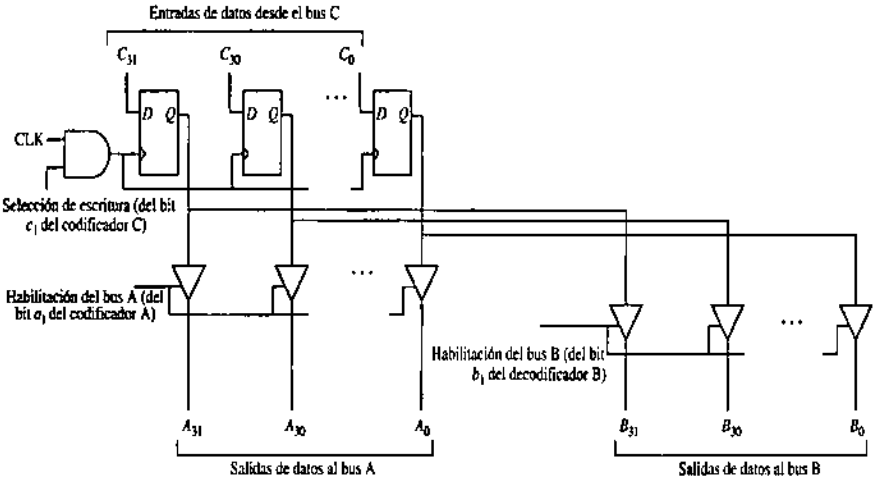


Figura 6.8 • Diseño del registro `%r1`.

La entrada CLK al registro `%r1` se introduce en una compuerta Y junto con la línea de selección correspondiente (c_1) del decodificador C. Esto asegura que `%r1` solo cambia cuando la sección de control así lo determina. Las entradas de datos a `%r1` se toman directamente de las líneas correspondientes desde el bus C. Las salidas se escriben sobre las líneas correspondientes de los buses A y B a través de compuertas *buffer* de tres estados, las que se encuentran "desconectadas eléctricamente" salvo cuando sus entradas de habilitación toman el valor 1. Las salidas de los buffers se transfieren hacia los buses A y B, por medio de las salidas a_1 y b_1 de los decodificadores A y B respectivamente. Si ni a_1 ni b_1 se encuentran en su nivel alto (equivalente a un 1 lógico), las salidas de `%r1` se desconectan eléctricamente de ambos buses A y B debido a que los *buffers* correspondientes están inhibidos.

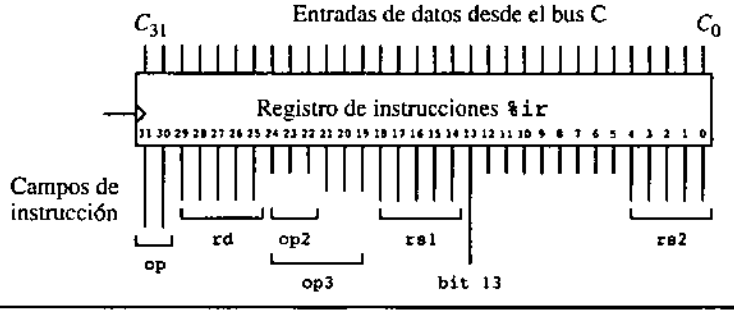


Figura 6.9 • Salidas hacia la unidad de control desde el registro `%ir`.

Los demás registros adoptan un esquema similar, salvo por algunas excepciones. El registro `%r0` siempre contiene un 0, el que no puede modificarse. Por consiguiente, el registro `%r0` no tiene entradas desde el bus C, ni tampoco desde el decodificador C y, por ende, no requiere *flip flops* (véase el problema 6.11). El registro `%ir` tiene salidas adicionales que corresponden a los campos `rd`, `rs1`, `rs2`, `op`, `op2`, `op3` y bit 13 de las respectivas instrucciones, según se muestra en la figura 6.9. La unidad de control utiliza estas salidas en la interpretación de las instrucciones, tal como se verá en la sección 6.2.4. El contador de programa solo puede contener valores que sean múltiplos de 4, por lo que los dos bits menos significativos de `%pc` pueden ser conectados eléctricamente a cero.

Los decodificadores A, B y C de la figura 6.3 simplifican la selección de los registros. Las entradas de los decodificadores, de seis bits, seleccionan un único registro para cada uno de los buses A, B y C. Existen $2^6 = 64$ posibles salidas desde los decodificadores, pero solo hay 38 registros de datos. El índice asignado a cada registro (en base 10 a la izquierda de cada uno de ellos) en la figura 6.3 indica el valor que debe colocarse en las entradas del decodificador para seleccionar el registro correspondiente. La salida 0 del decodificador C no se utiliza debido a que el registro `%r0` no puede ser escrito. Los índices mayores a 37 no se corresponden con ningún registro y pueden utilizarse cuando no se requiere conectar registro alguno a los buses.

6.2.2 La sección de control

La figura 6.10 muestra la totalidad de la arquitectura microprogramada de ARC. La figura ilustra el trayecto de datos, la unidad de control y las conexiones entre ellas. En el corazón de la unidad de control, una memoria de lectura (ROM) de 2048 palabras de 41 bits contiene los valores de todas las líneas que deben controlarse para implementar cada instrucción a nivel del usuario. En este contexto, la memoria de lectura suele considerarse como una **memoria de control** (*control store*). Cada palabra de 41 bits es una **microinstrucción**. La unidad de control es la responsable de la búsqueda de las microinstrucciones y de su ejecución, en forma bastante similar a la forma en que se buscan y ejecutan las instrucciones de ARC a nivel del usuario. La ejecución de microinstrucciones se controla a través del registro de instrucciones de microprograma (MIR, *microprogram instruction register*), del registro de estado `%psr` y de un mecanismo que permite determinar cuál es la siguiente microinstrucción a ejecutar, formado por la unidad de saltos de control (CBL, *Control Branch Logic*) y el multiplexor de direcciones de la memoria de control. No se requiere contador de programa para almacenar la dirección de la próxima instrucción del microprograma, dado que la misma se recalcula en cada ciclo de reloj, lo que significa que no debe almacenarse para futuros ciclos.

Cuando la microarquitectura inicia la operación (por ejemplo, en el momento del encendido), un circuito de inicialización (que no se muestra en la figura) coloca la micropalabra de la dirección 0 de la memoria de control en el registro de instrucciones de microprograma, para su ejecución. A partir de ese punto, se seleccionan las micropalabras a

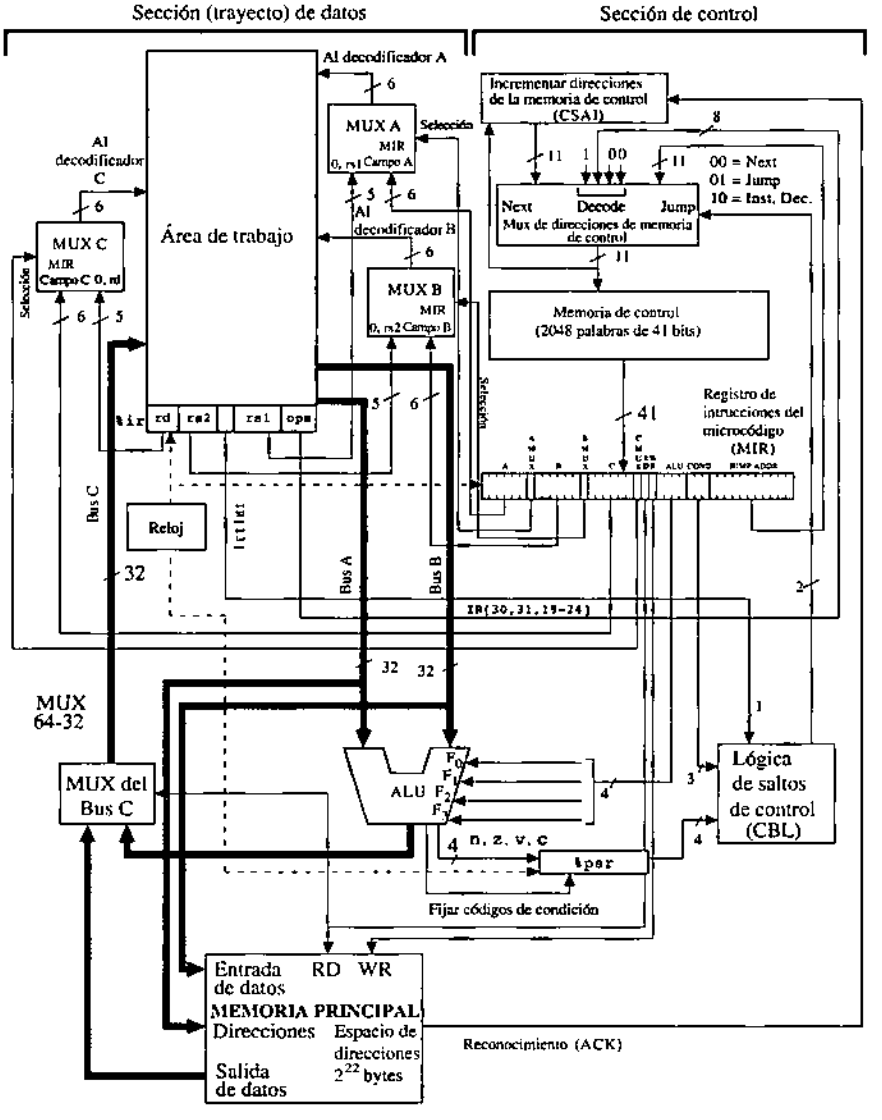


Figura 6.10 • La microarquitectura de ARC.

ejecutar desde alguna de las entradas Next, Decode o Jump del multiplexor de direcciones de la memoria de control, sobre la base de los valores que adoptan el campo COND del registro MIR de instrucciones de microprograma y la salida de la lógica de saltos de control. Luego de colocar cada palabra en el registro MIR, el trayecto de datos realiza las operaciones requeridas por los valores que adopten los diferentes campos del mismo registro. El proceso se detalla en el análisis siguiente.

Cada palabra de 41 bits comprende 11 campos distintos, de acuerdo con lo que se muestra en la figura 6.11. A partir de la izquierda, el campo A determina cuál es el registro del trayecto de datos cuyo contenido debe colocarse sobre el bus A. Los direcciona-

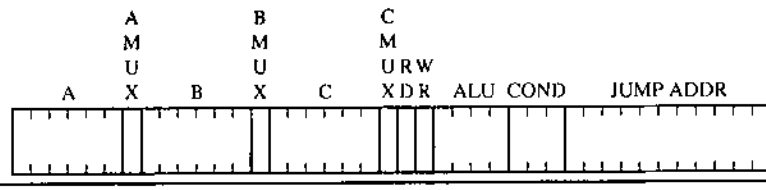


Figura 6.11 • El formato de la palabra de microcódigo.

mientos binarios de los registros se corresponden con las representaciones binarias de los índices decimales indicados en la figura 6.3 (000000-100101). El campo AMUX selecciona si el decodificador A obtiene su entrada desde el campo A del registro MIR (AMUX = 0) o desde el campo `rs1` del registro `%ir` (AMUX = 1).

En forma similar, el campo B determina cuál de los registros del trayecto de datos debe colocar su contenido sobre el bus B. El campo BMUX determina si el decodificador B obtiene sus entradas desde el campo B del MIR (BMUX = 0) o desde el campo `rs2` de `%ir` (BMUX = 1). El campo C determina en cuál de los registros del trayecto de datos se almacenará el dato transferido a través del bus C. El campo CMUX elige si la entrada al decodificador C se obtiene desde el campo C del MIR (CMUX = 0) o desde el campo `rd` de `%ir` (CMUX = 1). Dado que el contenido del registro `%r0` no puede modificarse, puede usarse el patrón binario 000000 cuando no se requiere modificar el contenido de ningún registro.

Las líneas RD y WR determinan, respectivamente, si se debe leer o escribir en memoria. Se realiza una lectura cuando RD = 1, en tanto que se realiza una escritura de memoria cuando WR = 1. Los campos RD y WR no pueden valer 1 en forma simultánea, pero sí pueden ser simultáneamente 0 si no se va a llevar a cabo una operación ni de lectura ni de escritura de memoria. Tanto para las operaciones de lectura como de escritura, la dirección de memoria se toma directamente desde el bus A. El dato que se ingresa en la memoria se toma desde el bus B y la salida de datos desde la memoria se coloca en el bus C. La línea RD controla el multiplexor de 64 a 32 bits del bus C, el que determina si el bus C se carga desde la memoria (RD = 1) o desde la unidad aritmético-lógica (RD = 0).

El campo denominado ALU determina cuál de las operaciones aritmético-lógicas se ejecuta de acuerdo con los valores establecidos en la figura 6.4. El campo ALU puede adoptar 16 valores diferentes, todos los cuales corresponden a operaciones aritmético-lógicas válidas. Esto implica que no hay forma de "apagar" la unidad aritmético-lógica cuando no se la necesita, tal como cuando se realiza una lectura o una escritura en memoria. En estos casos debe elegirse una operación de la unidad aritmético-lógica que no presente efectos colaterales indeseados. Por ejemplo, no sería apropiado presentarle a la unidad aritmético-lógica la instrucción ANDCC, que cambia los códigos de condición, pero sí podría aceptarse la operación AND, que no los afecta.

El campo COND (salto condicional) del formato de la microinstrucción hace que el **microcontrolador** rescate la micropalabra siguiente, ya sea desde la posición siguiente en la memoria de control, o desde la posición indicada en el campo JUMP ADDR del registro MIR, o bien desde los bits del código de operación almacenado en `%ir`. El campo COND

se interpreta de acuerdo con la tabla indicada en la figura 6.12. Si el campo COND vale 000, no hay salto alguno, y se utiliza la entrada Next del multiplexor de direcciones de la memoria de control. La entrada Next mencionada se transfiere al circuito destinado a incrementar las direcciones de la memoria de control (CSAI, *Control Store Address Incrementer*) de la figura 6.10, el que incrementa en 1 la salida actual del multiplexor de direcciones. Si el campo COND vale 001, 010, 011, 100 o 101, se procede a realizar un salto condicional a la posición de la memoria de control indicada en el campo JUMP ADDR, de acuerdo con el valor de las banderas `n`, `z`, `v` o `c`, o del bit 13 de `%ir`, respectivamente. La sintaxis "`IR[13]`" representa "el bit 13 del registro de instrucciones `%ir`". Si el campo COND vale 110, se produce un salto incondicional.

C_2 C_1 C_0	Operación
0 0 0	Usar NEXT ADDR
0 0 1	Usar JUMP ADDR if `n` = 1
0 1 0	Usar JUMP ADDR if `z` = 1
0 1 1	Usar JUMP ADDR if `v` = 1
1 0 0	Usar JUMP ADDR if `c` = 1
1 0 1	Usar JUMP ADDR if `IR[13]` = 1
1 1 0	Usar JUMP ADDR
1 1 1	DECODE

Figura 6.12 • Valores que adopta el campo COND de la palabra de microcódigo.

Durante la decodificación de una instrucción, el campo COND adopta el valor 111. Cuando el campo COND vale 111, la dirección de memoria de control que debe copiarse en el registro MIR no se toma ni desde la entrada Next del multiplexor de direcciones ni de la entrada Jump, sino desde una combinación de 11 bits creada mediante el agregado de un 1 a la izquierda de los bits 30 y 31 de `%ir` y del agregado de ceros a la derecha de los bits 19-24 del mismo registro `%ir`. El formato de direcciones de DECODE se ilustra en la figura 6.13. El objetivo de la utilización de este esquema de direccionamiento es el de permitir que la instrucción sea decodificada en un solo paso, a través de saltos a diferentes ubicaciones definidas por los valores que adoptan los campos `op`, `op2` y `op3` de la instrucción.

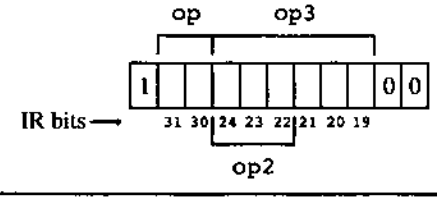


Figura 6.13 • El formato de DECODE para generar la dirección en una microinstrucción.

Finalmente, el campo JUMP ADDR aparece en los 11 bits menos significativos del formato de las micropalabras. Existen 2^{11} micropalabras en la memoria de control, por lo que se requieren 11 bits de direccionamiento para poder acceder a cualquiera de las posiciones de la memoria de control.

6.2.3 Temporización

La microarquitectura opera sobre un ciclo de reloj de dos fases, en el que las secciones maestras de todos los registros cambian en el flanco positivo del reloj, en tanto que las secciones esclavas de los registros cambian en el flanco negativo del mismo, tal como se indica en la figura 6.14. Todos los registros usan circuitos biestables tipo D de estructura maestro-esclavo activados en su flanco negativo, excepto por `%r0`, el que no requiere biestables. En el flanco negativo del reloj, la información almacenada en la sección maestra de cada *flip flop* se transfiere hacia la sección esclava del mismo. Esta operación deja la información disponible para las operaciones que involucran a la unidad aritmético-lógica. Cuando el reloj está en su estado bajo, se llevan a cabo las funciones ALU, MUX y CBL, las que deben haberse completado al momento del siguiente flanco positivo del reloj. Los registros acomodan sus valores cuando la señal de reloj se encuentra en su nivel alto, y entonces el proceso se repite.

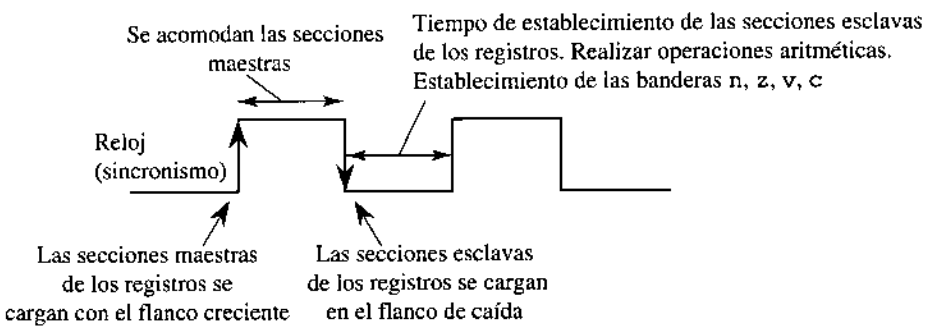


Figura 6.14 • Relaciones de tiempo para el funcionamiento de los registros.

6.2.4 El desarrollo del microprograma

En una arquitectura microprogramada, las instrucciones se interpretan desde el microprograma almacenado en la memoria de control. Suele hablarse de ***firmware*** cuando se hace mención al microprograma debido a que el mismo establece un puente entre el hardware y el software de la máquina. La arquitectura de la figura 6.10 requiere *firmware* para poder ejecutar las instrucciones de ARC. Esta sección describe una de las posibles codificaciones.

La figura 6.15 muestra una porción de un microprograma que implementa el ciclo de búsqueda y ejecución de ARC. En la memoria de control, cada microsentencia se alma-

cena en forma codificada (ceros y unos) en una única micropalabra. Con el objeto de simplificar la cuestión, el **lenguaje microensamblador** de la figura 6.15 se trata muy superficialmente, evitando el análisis de rótulos, directivas, etc., normalmente asociados con un lenguaje ensamblador completo. Cuando el microprograma es chico, no es demasiado complicado realizar a mano la transcripción al formato de 41 bits usado en la memoria de control; es más, con frecuencia se lo realiza de esta forma (como se hará en este caso) en vez de crear toda una batería de herramientas de software para efectuar la traducción de un programa simple.

```
Dirección  Sentencias operativas                          Comentario
     0: R[ir] ← AND(R[pc],R[pc]); READ;     / Leer una instrucción de ARC desde memoria principal
     1: DECODE;                             / Salto (256 posibilidades) condicionado al código de operación
     / sethi
  1152: R[rd] ← LSHIFT10(ir); GOTO 2047;    / Copiar el campo imm22 en el registro de destino
     / call
  1280: R[15] ← AND(R[pc],R[pc]);           / Guardar %pc en %15
  1281: R[temp0] ← ADD(R[ir],R[ir]);        / Desplazar el campo disp30 a izquierda
  1282: R[temp0] ← ADD(R[temp0],R[temp0]); / Desplazar nuevamente
  1283: R[pc] ← ADD(R[pc],R[temp0]);        / Salto a subrutina
        GOTO 0;
        / addcc
  1600: IF R[IR[13]] THEN GOTO 1602;        / El segundo operando origen está en modo inmediato?
  1601: R[rd] ← ADDCC(R[rs1],R[rs2]);       / Resolver ADDCC sobre registros origen
        GOTO 2047;
  1602: R[temp0] ← SEXT13(R[ir]);           / Obtener el campo simm13, con extensión de signo
  1603: R[rd] ← ADDCC(R[rs1],R[temp0]);     / Resolver ADDCC sobre operandos origen en registro/simm13
        GOTO 2047;                          /
        / andcc
  1604: IF R[IR[13]] THEN GOTO 1606;        / El segundo operando origen está en modo inmediato?
  1605: R[rd] ← ANDCC(R[rs1],R[rs2]);       / Resolver ANDCC sobre registros origen
        GOTO 2047;
  1606: R[temp0] ← SIMM13(R[ir]);           / Obtener el campo simm13
  1607: R[rd] ← ANDCC(R[rs1],R[temp0]);     / Resolver ANDCC sobre operandos origen en registro/simm13
        GOTO 2047;                          /
        / orcc
  1608: IF R[IR[13]] THEN GOTO 1610;        / El segundo operando origen está en modo inmediato?
  1609: R[rd] ← ORCC(R[rs1],R[rs2]);        / Resolver NORCC sobre registros origen
        GOTO 2047;
  1610: R[temp0] ← SIMM13(R[ir]);           / Obtener el campo simm13
  1611: R[rd] ← ORCC(R[rs1],R[temp0]);      / Resolver ORCC sobre operandos origen en registro/simm13
        GOTO 2047;
        / orncc
  1624: IF R[IR[13]] THEN GOTO 1626;        / El segundo operando origen está en modo inmediato?
  1625: R[rd] ← NORCC(R[rs1],R[rs2]);       / Resolver ORNCC sobre registros origen
        GOTO 2047;
  1626: R[temp0] ← SIMM13(R[ir]);           / Obtener el campo simm13
  1627: R[rd] ← NORCC(R[rs1],R[temp0]);     / Resolver NORCC sobre operandos origen en registro/simm13
        GOTO 2047;                          /
        / srl
  1688: IF R[IR[13]] THEN GOTO 1690;        / El segundo operando origen está en modo inmediato?
  1689: R[rd] ← SRL(R[rs1],R[rs2]);         / Resolver SRL sobre registros origen
        GOTO 2047;
  1690: R[temp0] ← SIMM13(R[ir]);           / Obtener el campo simm13
  1691: R[rd] ← SRL(R[rs1],R[temp0]);       / Resolver SRL sobre operandos origen en registro/simm13
        GOTO 2047;
        / jmpl
  1760: IF R[IR[13]] THEN GOTO 1762;        / El segundo operando origen está en modo inmediato?
  1761: R[pc] ← ADD(R[rs1],R[rs2]);         / Resolver ADD sobre operandos origen en registro/simm13
        GOTO 0;
```

Figura 6.15 • Microprograma parcial de ARC. Las micropalabras se muestran en secuencia lógica (no numérica).

```
1762: R[temp0] ← SEXT13(R[ir]);          / Obtener el campo simm13, con extensión de signo
1763: R[pc] ← ADD(R[rs1],R[temp0]);      / Resolver ADD sobre operandos origen en registro/simm13
      GOTO 0;
      / ld
1792: R[temp0] ← ADD(R[rs1],R[rs2]);     / Calcular dirección origen
      IF R[IR[13]] THEN GOTO 1794;
1793: R[rd] ← AND(R[temp0],R[temp0]);    / Colocar dirección origen sobre el bus A
      READ; GOTO 2047;
1794: R[temp0] ← SEXT13(R[ir]);          / Obtener el campo simm13 para la dirección origen
1795: R[temp0] ← ADD(R[rs1],R[temp0]);   / Calcular dirección de origen
      GOTO 1793;
      / st
1808: R[temp0] ← ADD(R[rs1],R[rs2]);     / Calcular dirección de destino
      IF R[IR[13]] THEN GOTO 1810;
1809: R[ir] ← RSHIFT5(R[ir]); GOTO 40;   / Mover el campo rd hacia la posición del campo rs2
  40: R[ir] ← RSHIFT5(R[ir]);            / desplazándolo 25 bits a la derecha.
  41: R[ir] ← RSHIFT5(R[ir]);
  42: R[ir] ← RSHIFT5(R[ir]);
  43: R[ir] ← RSHIFT5(R[ir]);
  44: R[0] ← AND(R[temp0], R[rs2]);      / Colocar la dirección de destino sobre el bus A y
      WRITE; GOTO 2047;                  /  el operando sobre el bus B
1810: R[temp0] ← SEXT13(R[ir]);          / Obtener el campo simm13 para calcular la dirección de destino
1811: R[temp0] ← ADD(R[rs1],R[temp0]);   / Calcular dirección de destino
      GOTO 1809;
      / Instrucciones de salto: ba, be, bcs, bvs, bneg
1088: GOTO 2;                            / Árbol de decodificación para saltos
   2: R[temp0] ← LSHIFT10(R[ir]);        / Extender el signo de los 22 bits menos
   3: R[temp0] ← RSHIFT5(R[temp0]);      /  significativos de %temp0, desplazando
   4: R[temp0] ← RSHIFT5(R[temp0]);      /  primero 10 bits a izquierda, luego 10 bits a derecha
   5: R[ir] ← RSHIFT5(R[ir]);            / La extensión de signo se realiza a través de RSHIFT5
   6: R[ir] ← RSHIFT5(R[ir]);            / Mover el campo COND a IR[13] utilizando tres veces RSHIFT5
   7: R[ir] ← RSHIFT5(R[ir]);            / La extensión de signo no afecta la operación
   8: IF R[IR[13]] THEN GOTO 12;         / ¿La instrucción es ba?
      R[ir] ← ADD(R[ir],R[ir]);
   9: IF R[IR[13]] THEN GOTO 13;         / ¿La instrucción no es be?
      R[ir] ← ADD(R[ir],R[ir]);
  10: IF Z THEN GOTO 12;                 / Ejecutar be
      R[ir] ← ADD(R[ir],R[ir]);
  11: GOTO 2047;                         / El salto indicado por be no se ejecuta
  12: R[pc] ← ADD(R[pc],R[temp0]);       / Ejecutar el salto
      GOTO 0;
  13: IF R[IR[13]] THEN GOTO 16;         / ¿Es bcs?
      R[ir] ← ADD(R[ir],R[ir]);
  14: IF C THEN GOTO 12;                 / Ejecutar bcs
  15: GOTO 2047;                         / El salto indicado por bcs no se ejecuta
  16: IF R[IR[13]] THEN GOTO 19;         / ¿Es bvs?
  17: IF N THEN GOTO 12;                 / Ejecutar bneg
  18: GOTO 2047;                         / El salto indicado por bneg no se ejecuta
  19: IF V THEN GOTO 12;                 / Ejecutar bvs
  20: GOTO 2047;                         / El salto indicado por bvs no se ejecuta
2047: R[pc] ← INCPC(R[pc]); GOTO 0;      / Incrementar %pc y empezar de nuevo
```

Figura 6.15 • Continuación.

Si bien el lenguaje microensamblador es en realidad un lenguaje simbólico, no es el mismo tipo de lenguaje ensamblador analizado en el capítulo 4. El lenguaje ensamblador de ARC es visible al usuario y se utiliza para la codificación de programas genéricos. Este lenguaje microensamblador se utiliza para la codificación del *firmware* y no es accesible al usuario. El único propósito del *firmware* es el de interpretar el conjunto de instrucciones visible por el usuario. Una modificación en el conjunto de instrucciones del procesador involucra cambios en el *firmware*, en tanto que una modificación a nivel del software escrito por el usuario no influye sobre el *firmware*.

Cada sentencia del microprograma de la figura 6.15 está antecedida por un número decimal que indica la dirección de la micropalabra correspondiente en la memoria de control de 2048 palabras. La dirección termina en un separador representado por dos puntos (:).

A continuación de la dirección aparece el campo que contiene las sentencias de operación, el que finaliza con un punto y coma. Luego del campo de operación se admite un campo opcional de comentarios, el que se inicia con una barra inclinada (/). El campo de comentarios termina al finalizar el renglón. Se admite más de una operación por línea, siempre que todas las operaciones puedan realizarse en un único ciclo de instrucción. Las operaciones aritmético-lógicas se obtienen de la figura 6.4; además existen algunas otras, las que se verán más adelante. Nótese que las 65 sentencias están planteadas en una secuencia lógica, más que numérica.

Antes de iniciada la ejecución por parte del microprograma, el contador de programa se inicializó con la dirección de comienzo del programa cargado en la memoria principal. Esto puede ocurrir como resultado de una secuencia de inicialización en el momento del encendido de la computadora, o bien desde el sistema operativo durante el funcionamiento normal del sistema.

La primera tarea en la ejecución de un programa a nivel del usuario es la de cargar la instrucción a la que apunta el contador de programa, desde la memoria principal hacia el registro de instrucciones IR. Obsérvese en la figura 6.10 que las líneas de dirección a la memoria principal se toman desde el bus A. En la línea 0 del microprograma se carga el contador de programa en el bus A, y se genera una operación de lectura de la memoria. La notación "`R[x]`" representa el "registro x", expresión en la que `x` puede reemplazarse por alguno de los registros del trayecto de datos, de modo tal que "`R[1]`" representa al registro `%r1`, "`R[ir]`" representa al registro `%ir` y "`R[rs1]`" identifica al registro que aparece en el campo de cinco bits `rs1` de una instrucción (según la figura 6.2).

La expresión "`AND(R[pc],R[pc])`", interpretada en forma literal, simplemente realiza el producto lógico del contador de programa consigo mismo. En sentido lógico, esta operación no parece ser demasiado útil, pero puede ser útil el análisis de sus efectos secundarios. Con el objeto de colocar `%pc` en el bus A, hace falta elegir una operación de la unidad aritmético-lógica que utilice el bus pero que no afecte los códigos de condición. De la variedad de alternativas posibles, se elige arbitrariamente la operación correspondiente al producto lógico AND. Debe notarse que el resultado del producto lógico se descarta debido a que el multiplexor del bus C de la figura 6.10 solo permite que la información que sale de la memoria principal pase al bus C durante una operación de lectura.

Una operación de lectura requiere normalmente más tiempo para completar su ejecución que el tiempo requerido para la ejecución de una microinstrucción. El tiempo de acceso a la memoria principal varía según la organización de la memoria, tal como se analizará en el capítulo 7. Con el objeto de considerar las variaciones en los tiempos de acceso de la memoria, el circuito que incrementa las direcciones de la memoria de control (CSAI) no lleva a cabo el incremento de la dirección hasta que no se haya recibido una señal de reconocimiento (ACK) que indique que la memoria ha completado su operación.

El flujo de control dentro del microprograma se trasfiere a la sentencia cuyo número de identificación sea el siguiente en sentido creciente, salvo que se detecte una operación de GOTO o de DECODE. En tal caso, en el ciclo siguiente se carga el MIR con la micro-

palabra 1 (línea 1). Nótese que algunas de las sentencias de microcódigo de la figura 6.15 ocupan más de un renglón de texto en la figura, pero son parte de una misma microinstrucción. Véanse, por ejemplo, los casos de las líneas 1283 y 1601.

Ahora que la instrucción se encuentra en el registro de instrucción como resultado de la operación de lectura ejecutada en la línea 0, el paso siguiente es el de la decodificación de los campos correspondientes al código de operación. Esta operación se realiza ejecutando un salto hacia el microcódigo, de 256 posibilidades diferentes, según lo indica la palabra clave DECODE en la línea 1 del microprograma. La dirección del salto, de 11 bits, se construye, como ya se ha dicho, agregando un 1 a la izquierda de los bits 30 y 31 del contenido del registro de instrucción, seguido por los bits 19-24 del registro, seguidos por 00. Luego de decodificar los campos del código de operación, la ejecución del microcódigo continúa en función de cuál de las 15 instrucciones de ARC está siendo interpretada.

En un ejemplo de la forma de funcionamiento de la operación de decodificación, considérese la instrucción `addcc`. De acuerdo con el formato de instrucciones aritméticas de la figura 6.2, el campo `op` es 10 y el campo `op3` es 01 0000. Si se agrega un 1 a la izquierda de la palabra binaria correspondiente a `op`, seguido del valor binario de `op3`, y seguido por 00, la dirección de DECODE es 110 0100 0000 = $(1600)_{10}$. Esto significa que las microinstrucciones que interpretan la instrucción `addcc` comienzan en la posición 1600 de la memoria de control.

En la práctica, existe una cantidad de direcciones de DECODE que no deberían aparecer nunca. No hay instrucción aritmética que responda a un formato binario 111111 en el campo `op3`, pero en previsión de que esta situación se produjese, quizás debido a un programa errante, se deberá colocar en la dirección de DECODE 110 1111 1100 = $(1788)_{10}$ una rutina de microcódigo capaz de lidiar con la instrucción ilegal. Estas locaciones se han dejado en blanco en el microprograma de la figura 6.15.

Las instrucciones de formato `SETHI/Branch` y `Call` no tienen campo de operandos `op3`. Los formatos `SETHI/Branch` solo poseen campos `op` y `op2`, en tanto que `Call` solo ofrece un campo `op`. Con el objeto de mantener un mecanismo simple de decodificación, pueden crearse entradas duplicadas en la memoria de control. Considerando el formato `SETHI`, si se sigue la regla determinada para obtener la dirección de DECODE, esta dirección deberá tener un 1 en su bit más significativo, seguido por 00 en el campo `op`, seguido a su vez por 100, que identifica a `SETHI` en las posiciones 19-21 de la palabra, seguidos por los bits de las posiciones 22-24 del registro de instrucción, finalmente seguidos por 00. Se obtiene como resultado la palabra binaria 100100xxx00, en la que xxx puede adoptar cualquiera de los valores posibles dependiendo del campo `imm22`. Los tres bits xxx pueden adoptar 8 valores diferentes, por lo que será necesario duplicar códigos SETHI en las posiciones de memoria identificadas como 100 1000 **0000**, 100 1000 **0100**, 100 1000 **1000**, 100 1000 **1100**, 100 1001 **0000**, 100 1001 **0100**, 100 1001 **1000** y 100 1001 **1100**. Las direcciones de DECODE en los formatos de bifurcación (Branch) y llamado (Call) se construyen en posiciones duplicadas en forma similar. La figura 6.15 representa solo la versión de cada conjunto de códigos duplicados ubicada en la dirección numéricamente más baja.

Si bien este método de decodificación es simple y rápido, se pierde gran cantidad de memoria de control. Como solución alternativa que desperdicia mucho menos espacio en la memoria de control, se puede modificar el decodificador de la memoria de control de modo tal que todos los formatos de salto de SETHI apunten a la misma posición. El mismo criterio puede usarse para resolver las instrucciones con formato de bifurcación (Branch) y llamado (Call). En la microarquitectura del ejemplo, se prefiere la solución más simple aun cuando haya que pagar el precio de una memoria de control más grande.

Se analizará ahora la forma en que se implementa la instrucción `ld`. El microprograma comienza en la posición 0, aun cuando en este punto no sabe cuál es el código de operación al que apunta el contador de programa en la memoria principal. La línea 0 del microprograma comienza la operación de lectura, tal como lo indica la palabra clave READ, con lo que se produce la carga de una instrucción desde la dirección de memoria principal, a la que apunta el contador de programa, hacia el registro de instrucciones. Para este caso, se supondrá que el registro de instrucciones contiene el formato de 32 bits siguiente:

```
11      00010       000000        00101     1       0 0000 0101 0000
op        rd          op3          rs1      i              simm13
```

Este formato corresponde a la traducción del código simbólico ARC `ld %r5 + 80, %r2`. En consecuencia, en la línea 1 se realiza el salto a la dirección $(111\ 0000\ 0000)_2 = (1792)_{10}$.

En la línea 1792 se inicia la ejecución de la instrucción `ld`. En esta línea, 1792, se analiza el valor del bit de direccionamiento inmediato `i`. En este ejemplo, `i = 1` por lo que se transfiere el control a la palabra ubicada en 1794. Si el valor de `i` hubiese sido 0, el control se hubiera transferido a la siguiente palabra, la que en este caso es 1793. La línea 1792 suma los registros indicados en los campos `rs1` y `rs2` de la instrucción, anticipándose a la forma no inmediata de la instrucción `ld`, lo que solo tiene sentido si `i = 0`. Dado que este no es el caso, el resultado almacenado en `%temp0` se descarta al momento de la transferencia de control a la línea 1794. Esta situación no genera ninguna señalización con respecto a tiempos, ni produce tampoco efectos secundarios indeseados, dado que ADD no modifica los códigos de condición.

En la palabra 1794 del microcódigo, se rescata el campo `simm13` (utilizando extensión de signos, según lo indica la operación SEXT13), el que, en la palabra 1795, se suma con el registro indicado en el campo `rs1`. Se transfiere luego el control a la palabra 1793, en la que se produce la operación de lectura. Se avanza ahora a la palabra 2047, en la que se incrementa el contador de programa con anticipación a la lectura de la siguiente instrucción a ser rescatada desde la memoria principal. Dado que las instrucciones ocupan cuatro bytes y deben estar almacenadas en posiciones de memoria cuyo valor sea múltiplo de cuatro, el contador de programa se incrementa en cuatro unidades. Se devuelve el control a la línea 0 del microcódigo, en la que se repite el proceso. Se requiere así un total de siete microinstrucciones con el objeto de interpretar la instrucción `ld`. A continuación, se repite dicho conjunto de microinstrucciones.

```
   0: R[ir] ← AND(R[pc],R[pc]); READ;    / Leer una instrucción de ARC desde memoria principal.
   1: DECODE;                            / Salto (256 posibilidades) condicionado al código de operación
1792: R[temp0] ← ADD(R[rs1],R[rs2]);     / Calcular dirección origen
      IF IR[13] THEN GOTO 1794;
1794: R[temp0] ← SEXT13(R[ir]);          / Obtener campo simm13 para la dirección origen
1795: R[temp0] ← ADD(R[rs1],R[temp0]);   Calcular dirección origen
      GOTO 1793;
1793: R[rd] ← AND(R[temp0],R[temp0]);    / Colocar dirección origen sobre el bus A
      READ; GOTO 2047;
2047: R[pc] ← INCPC(R[pc]); GOTO 0;      / Incrementar %pc y empezar de nuevo
```

Las instrucciones restantes, excepto en el caso de las instrucciones de salto, se interpretan en forma similar a la interpretación que se acaba de realizar para la instrucción `ld`. En las instrucciones de salto se requiere alguna decodificación adicional debido a que el tipo de salto queda determinado por el campo COND del formato de las instrucciones de salto (bits 25-28), campo que no se utiliza durante la operación de decodificación. La solución que aquí se utiliza consiste en desplazar los bits del campo COND hacia `IR[13]` de a uno por vez, saltando luego a distintas locaciones del microcódigo según el formato binario de COND.

Para las instrucciones de salto, la operación DECODE de la línea 1 del microprograma transfiere el control a la línea 1088. Estas instrucciones de salto requieren más espacio que las cuatro palabras admitidas para cada instrucción, por lo que desde la línea 1088 se devuelve el control a la línea 2, en la que se inicia una sección suficientemente grande de memoria disponible para almacenamiento de control.

Las líneas 2-4 rescatan los 22 bits que corresponden al desplazamiento del salto, colocan en cero los 10 bits más significativos y almacenan el valor obtenido en el registro `%temp0`. Esto se logra corriendo `%ir` en 10 posiciones a la izquierda, almacenándolo en `%temp0`, y, por último, corriendo el resultado nuevamente en 10 bits, ahora a la derecha. (Debe notarse que el desplazamiento puede ser negativo, por lo que las operaciones sobre el mismo se realizan con extensión de signo. RSHIFT5 implementa la extensión del signo.)* Las líneas 5-7 desplazan `%ir` en 15 bits a la derecha de modo de alinear el bit más significativo del campo COND (`IR[28]`) con la posición (`IR[13]`), lo que permite que una operación de `Jump on IR[13] = 1` analice el valor de cada bit. En forma alternativa, se podría desplazar el campo COND de `IR[31]` de a un bit por vez y utilizar la condición `Jump on n` para verificar cada bit. (Nótese la existencia de un ligero error en la forma en que se actualiza el contador de programa en la línea 12. El problema 6.21 puede dar la explicación.)

El proceso de decodificación del salto se inicia en la línea 8. El mismo se resume en la figura 6.16. Si `IR[28]`, que se encuentra ahora en `IR[13]`, vale 1, la instrucción es `ba`, la que se ejecuta en la línea 12. Nótese que el control vuelve a la línea 0, en lugar de saltar a la línea 2047, para evitar el doble incremento del contador de programa en la misma instrucción.

---

* *N. de T.*: Debe observarse con cuidado la utilización del mismo término (desplazamiento) para traducir tanto la operación realizada (*shift*, en inglés) cuanto la designación del dato (*offset*) que, en una instrucción de salto, identifica el destino de dicho salto.

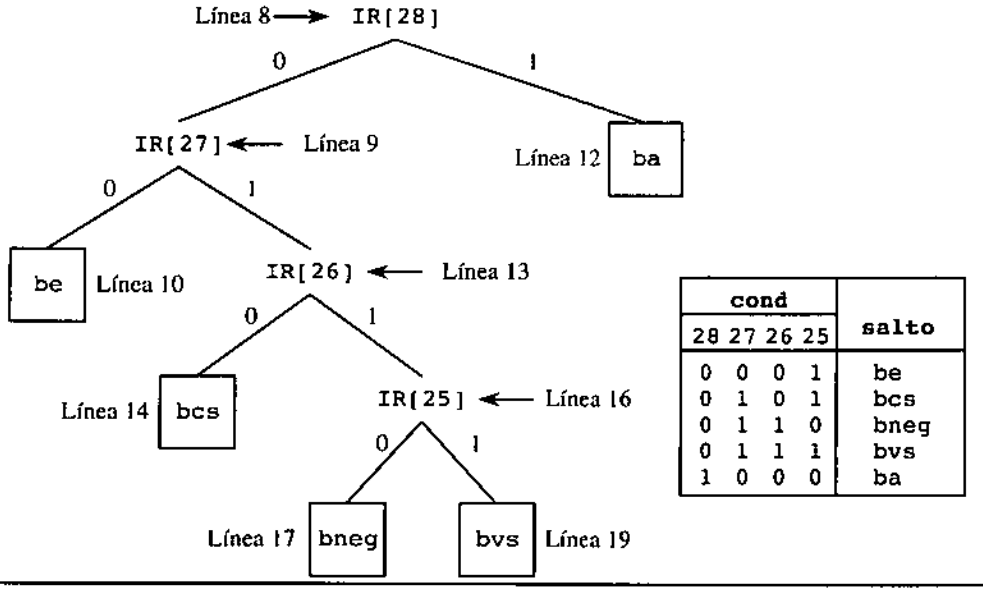


Figura 6.16 • Árbol de decodificación para las instrucciones de salto y sus correspondientes líneas de microprograma.

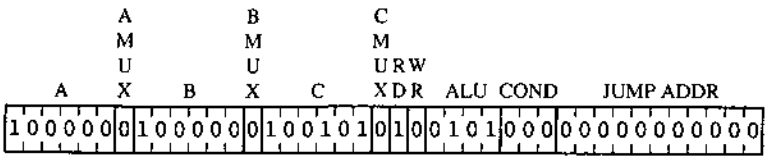
Si `IR[28] = 0`, se desplaza `%ir` un bit a la izquierda a través de la suma del registro consigo mismo, de modo de alinear `IR[27]` en la posición `IR[13]`. El bit `IR[27]` se verifica en la línea 9. Si su valor es cero, se ejecuta la instrucción be en la línea 10; en caso contrario, se desplaza `%ir` a la izquierda, verificando el estado del bit `IR[26]` en la línea 13. Las instrucciones de salto restantes se interpretan en forma similar.

Traducción del lenguaje microensamblador

Un microprograma escrito en lenguaje microensamblador debe traducirse al código objeto binario antes de ser almacenado en la memoria de control, tal como se debe traducir un programa escrito en lenguaje simbólico antes de almacenarlo en memoria principal en la forma de un programa objeto binario. Cada línea del microprograma de ARC corresponde exactamente a una palabra de la memoria de control, y no existen en el programa referencias hacia delante sin identificar. De tal manera, el microprograma ARC se puede ensamblar línea por línea en un único paso. Considérese el ensamble de la línea 0 del microprograma que se muestra en la figura 6.15:

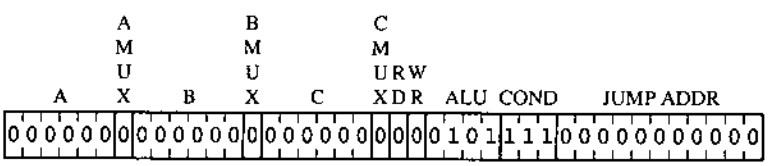
```
0:  R[ir]  ←AND  (R[pc],R[pc]);  READ;
```

Los campos de la micropalabra de 41 bits se pueden completar en la forma siguiente:



El producto lógico del contador de programa consigo mismo se inicia cargando el mencionado registro en ambos buses A y B, con lo que se transfiere una palabra a través de la unidad aritmético-lógica, sin modificarla. Los campos A y B tienen el formato correspondiente al contador de programa (32_{10} = 10 0000_2). Los campos AMUX y BMUX contienen ceros dado que las entradas a estos multiplexores se toman desde el registro MIR. El registro de destino de la operación de lectura es el registro de instrucciones IR, cuya identificación binaria corresponde a 37_{10} = 10 0101_2 en el campo C. El campo CMUX contiene un cero debido a que la entrada al CMUX se toma desde el MIR. Se requiere realizar una operación de lectura de memoria, por lo que el campo RD contiene un 1 en tanto que el campo WR contiene un 0. El campo ALU contiene 0101, lo que corresponde a la operación lógica AND. Nótese que los códigos de condición no se ven afectados, lo que sí ocurriría si se hubiese utilizado la instrucción ANDCC. El campo COND contiene 000 dado que el control se debe transferir a la micropalabra siguiente, por lo que la palabra binaria del campo JUMP ADDR no tiene importancia. Este campo, arbitrariamente, se completa con ceros.

La segunda micropalabra implementa el salto de 256 vías. En este caso, todo lo que importa es que en el campo COND de la operación DECODE aparezca el valor 111 y que no se afecte el contenido de registros, memoria o códigos de condición. El formato binario correspondiente es ahora:



La línea 1 podría admitir una cantidad de formatos diferentes y funcionar de todos modos. Por ejemplo, en los campos A, B y JUMP ADDR pueden aparecer distintas combinaciones binarias cuando se lleva a cabo una operación de DECODE. El uso de ceros para completar dichos campos es simplemente una decisión arbitraria. El campo ALU es 0101, correspondiente a la instrucción AND, lo que no afecta los códigos de condición. Pueden usarse, alternativamente, otros códigos de operación que no afecten los códigos de condición.

El resto del microprograma se traduce en forma similar. La figura 6.17 ilustra el microprograma traducido íntegramente, excepto en aquellos lugares en los que deberían aparecer códigos duplicados debido a instrucciones de salto o a códigos correspondientes a "instrucciones ilegales".

Ejemplo

Considérese el agregado de una instrucción a la implementación microprogramada del conjunto de instrucciones de ARC. Dicha instrucción, llamada `subcc`, resta su segundo operando del primero en notación de complemento a dos, utilizando el formato aritmético y un campo `op3` de 001100.

Se requiere modificar el microprograma para agregar la nueva instrucción. Se inicia el proceso determinando la dirección de comienzo de `subcc` en la memoria de control, para lo que se agrega un 1 a la izquierda del campo `op`, cuyo valor es 10, seguido por el campo `op3`, cuyo valor es 00 1100 y, finalmente, por 00. El resultado de esta construcción es la palabra binaria 110 0011 0000, que corresponde a la dirección $(1584)_{10}$ dentro de la memoria de control. Ahora, corresponde crear el código microensamblado, el que resulta similar al de

Dirección de almacenamiento	A MUX A	B MUX B	C MUX C	URW	MDR	ALU	COND	JUMP ADDR
0	1000000	1000000	1001010	110	0	0101	000	000000000000
1	0000000	0000000	0000000	000	0	0101	111	000000000000
1152	1001010	0000000	0000000	100	1	0101	110	111111111111
1280	1000000	1000000	0001111	000	0	0101	000	000000000000
1281	1001010	1001010	1000001	000	1	0000	000	000000000000
1282	1000010	1000010	1000010	000	1	0000	000	000000000000
1283	1000000	1000010	1000000	000	1	0000	110	000000000000
1600	0000000	0000000	0000000	000	0	0101	101	110010000110
1601	0000001	0000000	1000000	100	0	0011	110	111111111111
1602	1001010	0000000	1000010	000	1	1100	000	000000000000
1603	0000001	1000010	0000000	100	0	0011	110	111111111111
1604	0000000	0000000	0000000	000	0	0101	101	110010000110
1605	0000001	0000000	1000000	100	0	0000	110	111111111111
1606	1001010	0000000	1000010	000	1	1011	000	000000000000
1607	0000001	1000010	0000000	100	0	0000	110	111111111111
1608	0000000	0000000	0000000	000	0	0101	101	110010010110
1609	0000001	0000000	1000000	100	0	0001	110	111111111111
1610	1001010	0000000	1000010	000	1	1011	000	000000000000
1611	0000001	1000010	0000000	100	0	0001	110	111111111111
1624	0000000	0000000	0000000	000	0	0101	101	110010110110
1625	0000001	0000000	1000000	100	0	0010	110	111111111111
1626	1001010	0000000	1000010	000	1	1011	000	000000000000
1627	0000001	1000010	0000000	100	0	0010	110	111111111111
1688	0000000	0000000	0000000	000	0	0101	000	110100110110
1689	0000001	0000000	1000000	100	0	0100	110	111111111111
1690	1001010	0000000	1000010	000	1	1011	000	000000000000
1691	0000001	1000010	0000000	100	0	0100	110	111111111111
1760	0000000	0000000	0000000	000	0	0101	101	110111000110
1761	0000001	0000000	1000000	000	0	1000	110	000000000000
1762	1001010	0000000	1000010	000	1	1100	000	000000000000
1763	0000001	1000010	1000000	000	0	1000	110	000000000000
1792	0000001	0000000	1000010	000	1	1000	101	111000000010

Figura 6.17 • Microprograma ensamblado del subconjunto de instrucciones de ARC.

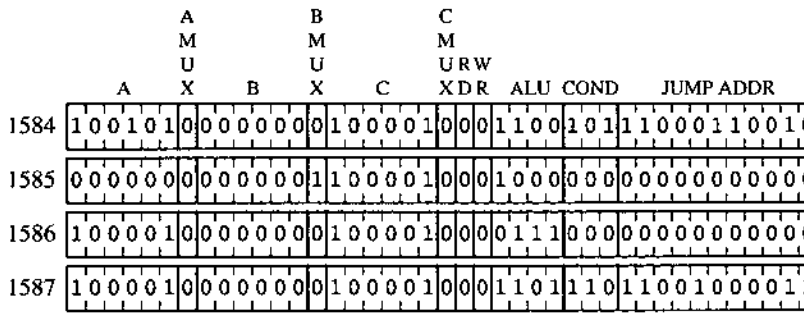
	A	AMUX	B	BMUX	C	CMUXDR		ALU	COND	JUMP ADDR
1793	100001	0	100001	0	000000	1	10	0101	110	11111111111
1794	100101	0	000000	0	100001	0	00	1100	000	00000000000
1795	000000	1	100001	0	100001	0	00	1000	110	11100000001
1808	000000	1	000000	1	100001	0	00	1000	101	11100010010
1809	100101	0	000000	0	100101	0	00	1111	110	00000101000
40	100101	0	000000	0	100101	0	00	1111	000	00000000000
41	100101	0	000000	0	100101	0	00	1111	000	00000000000
42	100101	0	000000	0	100101	0	00	1111	000	00000000000
43	100101	0	000000	0	100101	0	00	1111	000	00000000000
44	100001	0	000000	1	000000	0	01	0101	110	11111111111
1810	100101	0	000000	0	100001	0	00	1100	000	00000000000
1811	000000	1	100001	0	100001	0	00	1000	110	11100010001
1088	000000	0	000000	0	000000	0	00	0101	110	00000000010
2	100101	0	000000	0	100001	0	00	1010	000	00000000000
3	100001	0	000000	0	100001	0	00	1111	000	00000000000
4	100001	0	000000	0	100001	0	00	1111	000	00000000000
5	100101	0	000000	0	100101	0	00	1111	000	00000000000
6	100101	0	000000	0	100101	0	00	1111	000	00000000000
7	100101	0	000000	0	100101	0	00	1111	000	00000000000
8	100101	0	100100	0	100101	0	00	1000	101	00000001100
9	100101	0	100100	0	100101	0	00	1000	101	00000001101
10	100101	0	100100	0	100101	0	00	1000	010	00000001100
11	000000	0	000000	0	000000	0	00	0101	110	11111111111
12	100000	0	100001	0	100000	0	00	1000	110	00000000000
13	100101	0	100101	0	100101	0	00	1000	101	00000010000
14	000000	0	000000	0	000000	0	00	0101	100	00000001100
15	000000	0	000000	0	000000	0	00	0101	110	11111111111
16	000000	0	000000	0	000000	0	00	0101	101	00000010011
17	000000	0	000000	0	000000	0	00	0101	001	00000001100
18	000000	0	000000	0	000000	0	00	0101	110	11111111111
19	000000	0	000000	0	000000	0	00	0101	011	00000001100
20	000000	0	000000	0	000000	0	00	0101	110	11111111111
2047	100000	0	000000	0	100000	0	00	1110	110	00000000000

Figura 6.17 • Continuación.

la instrucción `addcc` almacenada en la posición 1600, con la excepción previsible de complementar a dos el sustraendo antes de realizar la operación. El sustraendo se complementa con la instrucción `NOR`, sumándole luego un 1 a través de la operación `INC`. Se completa la resta utilizando el código correspondiente a `addcc`. La codificación de `subcc` en microcódigo es la siguiente:

```
1584: R[temp0] ← SEXT13(R[ir]);          / Obtener el operando rs2
      IF IR[13] THEN GOTO 1586;          / El segundo operando está en modo inmediato
1585: R[temp0] ← R[rs2];                 / Obtener operando inmediato con extensión de signo
1586: R[temp0] ← NOR(R[temp0], R[0]);    / Obtener el complemento a uno del sustraendo
1587: R[temp0] ← INC(R[temp0]); GOTO 1603; / Obtener el complemento a dos del sustraendo
```

En tanto que el microcódigo binario correpondiente resulta ser:



6.2.5 Traps e interrupciones

Se define un ***trap*** como el procedimiento automático de llamada generado por el hardware como consecuencia de una condición excepcional que se produce durante la ejecución de un programa, por ejemplo una instrucción ilegal, un desborde por encima o por debajo de los límites de representación admisibles, una división por cero, etc. Cuando se produce un *trap*, se transfiere el control a un administrador de *traps*, rutina que es parte del sistema operativo. Este administrador podría, por ejemplo, imprimir un mensaje y finalizar el programa agresor.

Una forma de manejar *traps* puede consistir en modificar el microcódigo, posiblemente para verificar los bits de estado. Por ejemplo, puede verificarse el bit v para ver si se produjo un desborde. En ese caso, el microcódigo podría cargar en el contador de programa (si hubiese ocurrido un *trap*) la dirección de comienzo de la rutina de administración de estas situaciones.

Normalmente, existe una sección fija de memoria destinada a las direcciones de comienzo de los administradores de *trap*, en las que se almacena una única palabra para cada rutina de administración. Esta sección de la memoria conforma una **tabla de saltos** que transfiere el control a los administradores, tal como lo ilustra la figura 6.18. Se usa una tabla de saltos para permitir que la dirección absoluta de cada situación pueda incluirse en el microcódigo, en tanto que los destinos de los saltos pueden modificarse a nivel del usuario para manejar cada situación en forma diferente.

Un *trap* históricamente habitual es el que tiene que ver con las instrucciones de punto flotante, las que pueden ser **emuladas** por el sistema operativo si no han sido directamente implementadas en el hardware del sistema. Las instrucciones de punto flotante poseen sus propios códigos, pero, si no se han implementado en el hardware (esto significa que el microcódigo no las conoce), generarán un *trap* por instrucción ilegal en el momento en que se intente su ejecución. Cuando se produce una instrucción ilegal, se transfiere el control al administrador de instrucciones ilegales, el que verifica si el problema surgió a partir de una instrucción de punto flotante. De ser así, transfiere el control a la rutina de emulación de punto flotante adecuada, según la causa que produjo el *trap*. Si bien actualmente las unidades de cálculo en formato de punto flotante vienen incorporadas dentro de los procesadores integrados, este método se sigue utilizando en otras aplicaciones en

Dirección	Contenido	Administrador
	. . .	
60	JUMP TO 2000	Instrucción inválida
64	JUMP TO 3000	Desborde (por exceso)
68	JUMP TO 3600	Desborde (por debajo del mínimo)
72	JUMP TO 5224	División por cero
76	JUMP TO 4180	Disco
80	JUMP TO 5364	Impresora
84	JUMP TO 5908	Teclado
88	JUMP TO 6048	Temporizador
	. . .	

Figura 6.18 • Una tabla de saltos para las rutinas de atención de interrupciones y administradores de *traps*.

las que se extiende el conjunto de instrucciones con otras, como, por ejemplo, las extensiones gráficas de una arquitectura de programación.

Las interrupciones son situaciones similares que se producen luego de algún evento circuital considerado como **excepción**, como el accionamiento de una tecla en un teclado por parte del usuario, un llamado telefónico entrante en un módem, una falla de alimentación, una temperatura inapropiada para el funcionamiento normal, etc. Los *traps* son de naturaleza sincrónica con la ejecución de un programa, en tanto que las interrupciones son asincrónicas. Así, un *trap* se producirá siempre en el mismo punto de un mismo programa que se ejecuta con el mismo conjunto de datos, mientras que la ocurrencia de interrupciones es prácticamente impredecible.

Cuando se presiona una tecla en un teclado que funciona por medio de interrupciones, la electrónica del teclado activa una línea de interrupciones sobre el bus, tras lo cual la CPU activa una línea de acuse de recibo ni bien se encuentra lista para atender el llamado (aquí entra en juego el tema del **arbitraje del bus**, que se analiza en el capítulo 8, el cual tiene relación con la eventual existencia de más de un dispositivo interrumpiendo simultáneamente). El circuito del teclado se identifica ante la CPU como el responsable de la interrupción, por medio de la colocación de su **vector de interrupciones** en el bus de datos. La unidad de proceso coloca el contador de programa y el registro de estado en la pila. Se utiliza el vector de interrupción para acceder en forma indexada a la tabla de saltos, la que contiene las direcciones de comienzo de las diferentes rutinas de atención de las interrupciones.

Cuando se inicia la ejecución de alguna rutina de atención de una interrupción o de un *trap*, la misma procede a rescatar en la pila aquellos registros que piensa modificar, realiza su tarea, recupera los registros previamente almacenados y, luego, regresa al programa interrumpido. El proceso de retorno desde un *trap* es diferente al de retorno desde una subrutina, dado que el acceso a un *trap* difiere del llamado a una subrutina (debido a que también debe salvarse y luego rescatarse el registro `%psr`). En la arquitectura ARC se

utiliza la instrucción `rett` (véase el capítulo 8) para retornar desde una interrupción o desde un *trap*. Una interrupción puede llegar a interrumpir a otras interrupciones, por lo tanto, lo primero que puede requerirse de una rutina de atención de interrupciones es que eleve su prioridad (usando una instrucción especial en **modo supervisor**) para evitar la posterior aceptación de otras rutinas de menor prioridad.

6.2.6 Nanoprogramación

Si la memoria de control es ancha y tiene una gran reiteración de las mismas palabras, puede ahorrarse espacio de memoria de microprograma colocando una copia de cada palabra de microcódigo en un elemento de **nanoalmacenamiento**, usando la memoria de microprograma como índice a la memoria de nanocódigo. Por ejemplo, en el microprograma de la figura 6.15, las líneas 1281 y 1282 son iguales. Las líneas 3, 4 y 40 a 44 son iguales, y así en una cantidad de otras microinstrucciones, especialmente en los microcódigos duplicados correspondientes a los saltos y a las instrucciones ilegales.

La figura 6.19(a) ilustra los requerimientos de espacio de la memoria utilizada originalmente para el microcódigo. Consta de 2048 palabras, cada una de las cuales tiene 41 bits, dando como resultado una capacidad de 2048 x 41 = 83.968 bits. Supóngase que en toda la memoria hay 100 micropalabras únicas (el microprograma de la figura 6.15 está incompleto, por lo que no permite la medición directa de la cantidad de microcódigo único). La figura 6.19(b) ilustra una configuración que utiliza nanocódigo, en la que se logra ahorro de espacio si en la secuencia de microcódigo original existe una cantidad de patrones binarios que se repiten. Las micropalabras únicas (100 en este caso) forman un nanoprograma, el que se almacena en una memoria de lectura de 100 palabras de 41 bits cada una.

El microprograma accede ahora en forma indexada al nanocodigo. El microprograma tiene la misma cantidad de palabras independientemente de si se utiliza nanocódigo o no; pero, cuando se utiliza nanocódigo, en la memoria de control se almacenan punteros a aquel en lugar de las palabras de 41 bits. En este caso, la memoria de control tiene ahora 2048 palabras de un tamaño de $\log_2(100) = 7$ bits. La complejidad del área de almacenamiento, cuando se utiliza nanoalmacenamiento, es de 100 x 41 + 2048 x 7 = 18.436 bits, lo que representa un ahorro importante en la superficie con relación al planteo microcodificado original.

Para un valor pequeño de m y un valor grande de n, siendo m la longitud del nanoprograma, se puede imaginar un gran ahorro de memoria. Esto libera un área que puede utilizarse con algún otro objetivo, posiblemente para mejorar la eficiencia. No obstante, en lugar de acceder solo al microcódigo, ahora se debe acceder al microcódigo y, luego, al nanocódigo. La máquina funcionará más lentamente pero ocupando un espacio menor.

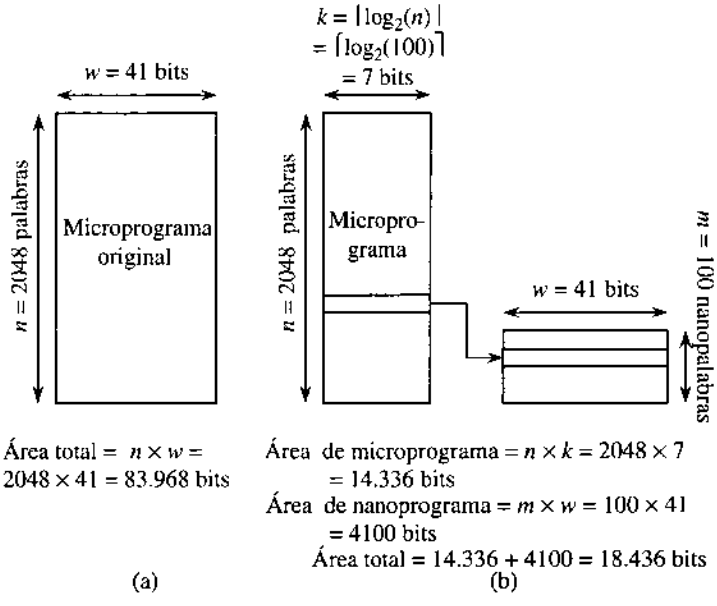


Figura 6.19 • (a) Microprogramación versus (b) nanoprogramación.

6.3 Control cableado

Una alternativa a la unidad de control microprogramada es el uso de un diseño **cableado**, en el que se realiza una implementación directa utilizando *flip flops* y compuertas lógicas, en lugar de usar un elemento de almacenamiento y un mecanismo de selección de micropalabras. Los pasos de un microprograma se reemplazan por los estados de una máquina de estados finitos.

Con el objeto de administrar la complejidad del diseño de una solución cableada, se suele utilizar un **lenguaje descriptor de hardware** (HDL, *Hardware Description Language*) con el objeto de representar la estructura de control. Un ejemplo de este tipo de lenguajes es **VHDL**, acrónimo de **VHSIC *Hardware Description Language*** (donde VHSIC, a su vez, es otro acrónimo de ***Very High Speed Integrated Circuit***). El lenguaje VHDL se utiliza para describir una arquitectura en un nivel muy elevado, pudiendo ser compilado hacia diseños de hardware a través de un proceso conocido como **compilación en silicio**. Para la unidad de control cableada a ser considerada en este capítulo, se utilizará, por ser más apropiado, un lenguaje HDL de menor nivel, el que habitualmente se conoce como **lenguaje de transferencia de registros** (RTL, *Register Transfer Language*).

El lenguaje HTL/RTL a ser definido en esta sección recuerda ligeramente al lenguaje **AHPL** (***A Hardware Programming Language***) desarrollado por F. J. Hill y G. R. Peterson. La idea general es la de expresar una secuencia de control como una serie de sentencias numeradas, cada una de las cuales puede, luego, traducirse directamente en un diseño de hardware. Cada sentencia consiste en una parte de datos y una de transferencia, como se observa a continuación:

```
5: A ← ADD (B,C);                  ! Sector de datos
GOTO {10 CONDITIONED ON IR[12]} ! Sector de control
```

La sentencia se rotula "5" con lo que se pretende indicar que viene precedida por la sentencia "4" y sucedida por la sentencia "6", a menos que se genere una transferencia de control que rompa la secuencia. La flecha a izquierda indica una transferencia de datos, en este caso hacia el registro A. La construcción "ADD (B,C)" indica que los registros B y C se suman en un circuito combinatorio. Los comentarios se inician con un signo de admiración (!) y se terminan en el final de la línea. La sentencia GOTO indica una transferencia de control. En este caso, se transfiere el control a la sentencia 10 si el bit 12 del registro IR es cierto; en caso contrario, se transfiere el control a la sentencia cuya numeración es la inmediata superior (6 en este caso).

Preámbulo

```
MODULE: MOD_4_COUNTER.
INPUTS: x.
OUTPUTS: Z[2].
MEMORY:
```

Sentencias

```
0: Z ← 0,0;
   GOTO {0 CONDITIONED ON x,
         1 CONDITIONED ON x̄}.
1: Z ← 0,1;
   GOTO {0 CONDITIONED ON x,
         2 CONDITIONED ON x̄}.
2: Z ← 1,0;
   GOTO {0 CONDITIONED ON x,
         3 CONDITIONED ON x̄}.
3: Z ← 1,1;
   GOTO 0.
```

Epílogo

```
END SEQUENCE.
END MOD_4_COUNTER.
```

Figura 6.20 • Secuencia de HDL para un contador módulo 4 con reinicialización.

La figura 6.20 muestra la descripción HDL de un contador módulo 4. El contador produce la secuencia de salidas 00, 01, 10, 11 y se repite en tanto la línea de entrada *x* valga 0. Si la línea de entrada pasa a valer 1, el contador vuelve al estado 0 al final del siguiente ciclo de reloj. La coma es el operador de concatenación y, por lo tanto, la sentencia "`Z ← 0,0;`" asigna el patrón `00` a la salida `Z` de dos bits.

La secuencia HDL está formada por tres secciones: el preámbulo, las sentencias numeradas y el epílogo. El preámbulo utiliza la palabra clave "MODULE" para designar al módulo y declara las entradas por medio de la palabra clave "INPUTS", las salidas con la palabra clave "OUTPUTS" y la cantidad de señales de ambas, así como cualquier necesidad adicional de almacenamiento, con la palabra clave "MEMORY" (ninguna en este ejemplo). Las sentencias numeradas aparecen luego del preámbulo. El epílogo completa la secuencia con la palabra clave "END SEQUENCE". La frase clave "END MOD_4 COUNTER" com-

pleta la descripción del módulo. Cualquier cosa que aparezca entre "END SEQUENCE" y "END MOD_4 COUNTER" ocurre en forma continua, independientemente de los números de sentencia. No hay sentencias de este tipo en el ejemplo.

Al traducir la descripción HDL en un diseño, el proceso puede descomponerse en partes separadas para las secciones de control y de datos. La sección de control maneja la forma de realizar las transiciones entre una sentencia y otra. La sección de datos está relacionada con la generación de las salidas y el cambio de los valores de cualquier elemento de memoria.

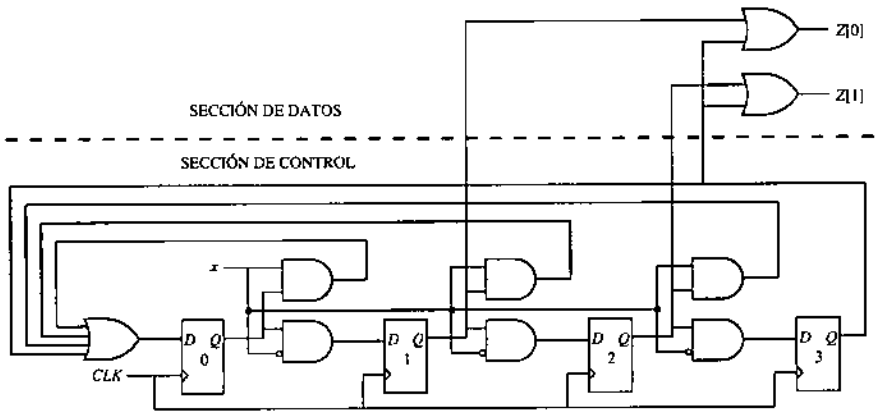


Figura 6.21 • Diseño lógico del contador módulo 4 descripto en HDL.

Se considerará primero la sección de control. Esta tiene cuatro sentencias numeradas, por lo que se utilizarán cuatro *flip flops*, uno para cada sentencia, como lo ilustra la figura 6.21. Se suele mencionar este tipo de diseño como **codificación "one hot"** debido a que, en cada momento, uno y solo uno de los *flip flops* contiene un valor cierto. Si bien cuatro estados pueden codificarse con solo dos *flip flops*, existen estudios que muestran que este tipo de codificación da por resultado, aproximadamente, un área de circuito similar a la de una implementación codificada en forma más densa. Más importante aún es el hecho de que las transferencias entre un estado y el siguiente son generalmente más simples y pueden ser implementadas con circuitos combinatorios sencillos, lo que a su vez implica una mayor velocidad de reloj para la solución de codificación "*one hot*" que para circuitos codificados más densamente.

Al diseñar la sección de control, se dibujan primero los *flip flops*, se aplican los rótulos que sean necesarios y se conectan las entradas de reloj. El paso siguiente consiste simplemente en recorrer en orden las sentencias numeradas y en agregar la lógica apropiada para lograr las transiciones. De la sentencia 0 existen dos posibles transiciones a las sentencias 0 o 1, condicionadas por x o su complemento, respectivamente. La salida del *flip flop* 0 se conecta entonces a las entradas de los *flip flops* 0 y 1, a través de compuertas Y que toman en cuenta el valor de la entrada x. Nótese que la compuerta Y que lleva al *flip flop* 1 tiene un círculo en una de sus entradas, indicando en forma simplificada que la entrada x debe ser negada por un inversor antes de ingresar a la compuerta Y.

Se utiliza una distribución similar de compuertas lógicas para las sentencias 1 y 2, en tanto que no se requiere lógica en la salida del *flip flop* 3 debido a que la sentencia 3 retorna incondicionalmente a la sentencia 1. Se ha completado la sección de control, la que puede funcionar ahora en forma autónoma. Sin embargo, no se tendrán salidas hasta tanto se implemente la sección de datos.

El diseño de la sección de datos, que se describe a continuación, resulta ser trivial en este caso. Los dos bits de la salida *Z* cambian en cada sentencia, por lo que no hay necesidad de condicionar la generación de la salida con el estado. Solo deben generarse los valores correctos de salida para cada una de las sentencias. El bit menos significativo de *Z* es 1 en las sentencias 1 y 3, por lo que las salidas de los correspondientes *flip flops* de control se suman lógicamente en una compuerta O para generar *Z*[0]. El bit más significativo de *Z* es cierto en las sentencias 2 y 3, por lo que nuevamente, utilizando una compuerta O, se procede a sumar las salidas de los correspondientes *flip flops* de control para producir *Z*[1]. El circuito completo del contador de módulo 4 se representa en la figura 6.21.

Puede utilizarse ahora el lenguaje HDL para describir la sección de control de la microarquitectura de ARC. No hay necesidad de diseñar la sección de datos, debido a que su aspecto ya ha sido definido en la figura 6.10. La sección de datos es la misma tanto en la solución microprogramada como en la cableada. Tal como en la solución microprogramada, las operaciones a ser realizadas por la unidad de control cableada son las que se describen a continuación:

1. Buscar la siguiente instrucción a ser ejecutada desde memoria.
2. Decodificar el código de operación.
3. Leer, si los hubiera, los operandos desde memoria principal o desde registros.
4. Ejecutar la instrucción y almacenar los resultados.
5. Volver al primer paso.

El microcódigo de la figura 6.15 puede servir como guía para entender lo que se requiere hacer. El primer paso consiste en buscar en la memoria principal la siguiente instrucción del nivel de usuario. La sentencia HDL que se muestra a continuación describe la operación:

```
0: ir ← AND (pc, pc); Read = 1
```

La estructura de esta sentencia es muy similar a la de la primera línea del microprograma, lo que no debe sorprender ya que se trata de la misma operación que debe ejecutarse sobre el mismo trayecto de datos.

Ahora que se ha procedido a la búsqueda de la instrucción, el paso siguiente es decodificar su código de operación. Aquí es donde entra en juego la potencia de una solución cableada. Dado que cada instrucción tiene un campo op, puede decodificarse ese campo primero y, luego, decodificar los campos op2, op3, cond, de acuerdo con lo que resulte apropiado para la instrucción.

La siguiente línea de la secuencia de control decodifica el campo op:

```
1: GOTO {2 CONDITIONED ON IR[31]×IR[30],   ! Formato para instrucciones de salto/sethi: op=00
         4 CONDITIONED ON IR[31]×IR[30],   ! Formato para instrucciones de llamada: op=01
         8 CONDITIONED ON IR[31]×IR[30],   ! Formato para instrucciones aritméticas: op=10
        10 CONDITIONED ON IR[31]×IR[30]}.  ! Formato para instrucciones de memoria: op=11
```

El símbolo de producto "x" indica una operación de producto lógico. Se transfiere así el control a alguna de las cuatro sentencias numeradas 2, 4, 8 o 10, de acuerdo con el formato binario del campo op.

La figura 6.22 muestra una descripción completa de la sección de control en lenguaje HDL. Puede llegar a requerirse una decodificación adicional según del valor del campo op. En la línea 4, correspondiente al formato Call, no se requiere decodificación adicional. La instrucción `call` se implementa en las sentencias 4–7, similares a las de la versión microprogramada.

En la sentencia 2, se requiere decodificación adicional sobre el campo op2, el que se verifica para determinar si la instrucción es `sethi` o es un salto. Dado que hay nada más que dos posibilidades, solo se requiere verificar un bit de op2 en la línea 2. La línea 3 implementa `sethi`, mientras que la línea 19 implementa las instrucciones de salto.

La línea 8 comienza con la sección de código correspondiente al formato de instrucciones aritméticas. La línea 8 rescata el segundo operando origen, que puede ser inmediato o directo y que puede requerir extensión de signo de 32 bits (en el caso de `addcc`) o no. La línea 9 implementa las instrucciones de formato aritmético, condicionadas por el campo op3. La función XNOR da un resultado cierto si sus argumentos son iguales; en caso contrario, su resultado es falso. Esto es útil cuando se realizan comparaciones.

En la línea 10 se inicia la sección de código que corresponde a las instrucciones de memoria. La línea 10 obtiene el segundo operando origen, el que puede ser tanto un registro como un operando inmediato. La línea 11 decodifica el campo op3. Dado que las únicas operaciones sobre memoria son `ld` y `st`, el campo op3 requiere la verificación de un solo bit (`IR[21]`). La línea 12 implementa la instrucción `ld`, en tanto que la instrucción `st` se implementa entre las líneas 13 y 18. Por último, la línea 20 incrementa el contador de programa y transfiere nuevamente el control a la primera sentencia.

Ahora que está definida la secuencia de control, el próximo paso es el diseño de la lógica de la sección de control. Dado que hay 21 sentencias, hay 21 *flip flops* en la sección de control, tal como lo ilustra la figura 6.23. Para cada uno de los 21 estados se genera una señal de control (CS_i), la que se usa en la sección de datos del controlador cableado.

En la figura 6.24, la sección de datos del controlador cableado genera las señales que controlan el trayecto de datos. Existen 27 compuertas O que corresponden a las 27 señales de control del trayecto de datos. (Haciendo referencia a la figura 6.10, existen 27 señales que se originan en la sección de control y que terminan en el trayecto de datos.) La señal AMUX se coloca en 1 nada más que en las líneas 9 y 11, correspondientes a las operaciones que colocan `rs1` en el bus A. La suma lógica de las señales CS_9 y CS_{11} genera la señal AMUX. En forma similar, `rd` aparece sobre el bus C en las líneas 3, 9 y 12, por lo que la suma lógica de las señales CS_3, CS_9 y CS_{12} genera la salida CMUX.

```
MODULE: ARC_CONTROL_UNIT.
INPUTS:
OUTPUTS: C, N, V, Z.  ! Fijadas por la unidad aritmético-lógica
MEMORY: R[16][32], pc[32], ir[32], temp0[32], temp1[32], temp2[32],
        temp3[32].

0: ir ← AND(pc, pc); Read ← 1;          ! Búsqueda de la instrucción
    ! Decodificar campo op
1: GOTO {2 CONDITIONED ON ir[31]×ir[30],  ! Formato brnch/sethi: op=00
         4 CONDITIONED ON ir[31]×ir[30],  ! Formato de llamada: op=01
         8 CONDITIONED ON ir[31]×ir[30],  ! Formato aritmético: op=10
        10 CONDITIONED ON ir[31]×ir[30]}. ! Formato memoria: op=11
    ! Decodifica campo op2
2: GOTO 19 CONDITIONED ON ir[24].         ! Si es formato de salto, pasar a línea 19
3: R[rd] ← ir[imm22];                     ! sethi
   GOTO 20.
4: R[15] ← AND(pc, pc).                   ! Llamada: salvar contador de programa en registro 15
5: temp0 ← ADD(ir, ir).                   ! Desplazar a izquierda el campo disp30
6: temp0 ← ADD(ir, ir).                   ! Desplazar nuevamente
7: pc ← ADD(pc, temp0); GOTO 0.           ! Salto a subrutina
    ! Colocar el segundo operando origen en temp0 si el formato es aritmético
8: temp0  ← { SEXT13(ir) CONDITIONED ON ir[13]×NOR(ir[19:22]),   ! addcc
  R[rs2] CONDITIONED ON ir[13]×NOR(ir[19:22]),                   ! addcc
  SIMM13(ir) CONDITIONED ON ir[13]×OR(ir[19:22]),  ! Instrucciones
  R[rs2] CONDITIONED ON ir[13]×OR(ir[19:22])}. ! aritméticas restantes
    ! Decodificar el campo op3 para el formato aritmético
9: R[rd] ← {
    ADDCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010000),  ! addcc
    ANDCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010001),  ! andcc
    ORCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010010),   ! orcc
    NORCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010110),  ! orncc
    SRL(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 100110),    ! srl
    ADD(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 111000)};   ! jmpl
    GOTO 20.
    ! Cargar en temp0 el segundo operando origen si el formato es de memoria
10: temp0  ← {SEXT13(ir) CONDITIONED ON ir[13],
              R[rs2] CONDITIONED ON ir[13]}.
11: temp0  ← ADD(R[rs1], temp0).
    ! Decodificar el campo op3 para el formato de memoria
    GOTO {12 CONDITIONED ON ir[21],                              ! ld
          13 CONDITIONED ON ir[21]}.                             ! st
12: R[rd] ← AND(temp0, temp0); Read ← 1; GOTO 20.
13: ir  ← RSHIFT5(ir).
14: ir  ← RSHIFT5(ir).
15: ir  ← RSHIFT5(ir).
16: ir  ← RSHIFT5(ir).
17: ir  ← RSHIFT5(ir).
18: r0  ← AND(temp0, R[rs2]); Write  ← 1; GOTO 20.
19: pc  ← {  ! Instrucciones de salto
    ADD(pc, temp0) CONDITIONED ON ir[28] + ir[28]×ir[27]×Z +
        ir[28]×ir[27]×ir[26]×C + ir[28]×ir[27]×ir[26]×ir[25]×N +
        ir[28]×ir[27]×ir[26]×ir[25]×V,
    INCPC(pc) CONDITIONED ON ir[28]×ir[27]×Z +
        ir[28]×ir[27]×ir[26]×C + ir[28]×ir[27]×ir[26]×ir[25]×N +
        ir[28]×ir[27]×ir[26]×ir[25]×V};
    GOTO 0.
20: pc ← INCPC(pc); GOTO 0.
END SEQUENCE.
END ARC_CONTROL_UNIT.
```

Figura 6.22 • Descripción HDL de la unidad de control de ARC.

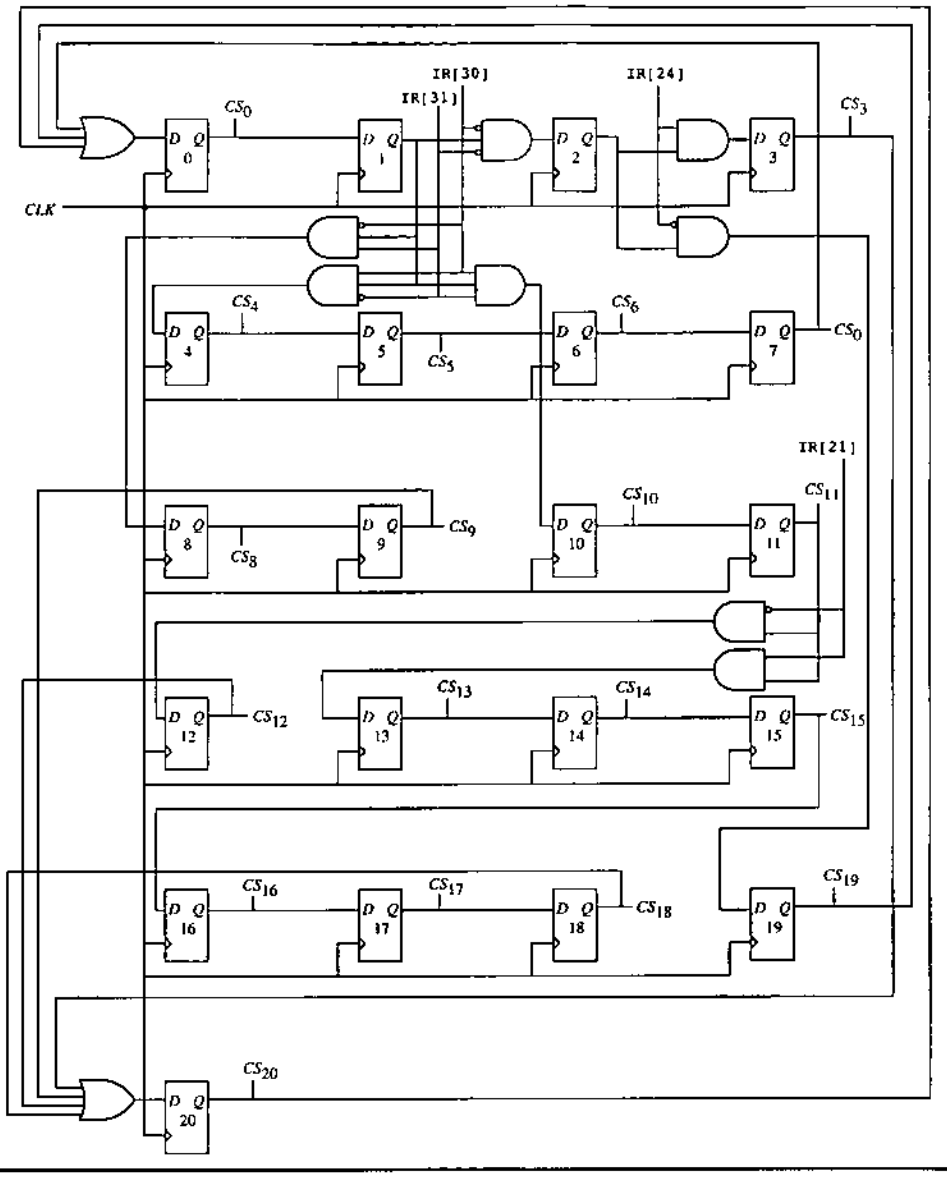


Figura 6.23 • La sección de control cableado en ARC: generación de las señales de control.

La señal BMUX es más compleja. Dado que `rs2` aparece sobre el bus B en las líneas 8, 10 y 18, se obtiene BMUX a través de la suma lógica de CS_8, CS_{10} y CS_{18}. No obstante, en la línea 8, BMUX se encuentra en 1 (indicando que `rs2` sale al bus B) solo si `IR[13] = 0` y si `IR[19:22]` se encuentran todos en cero (se trata de los cuatro bits menos significativos del patrón correspondiente a `op3` en la instrucción `addcc`: 010000). La figura ilustra la lógica correspondiente a este caso. Asimismo, en la línea 10, la señal BMUX vale 1 solo si `IR[13] = 0`. Nuevamente, la figura 6.24 ilustra la lógica correspondiente.

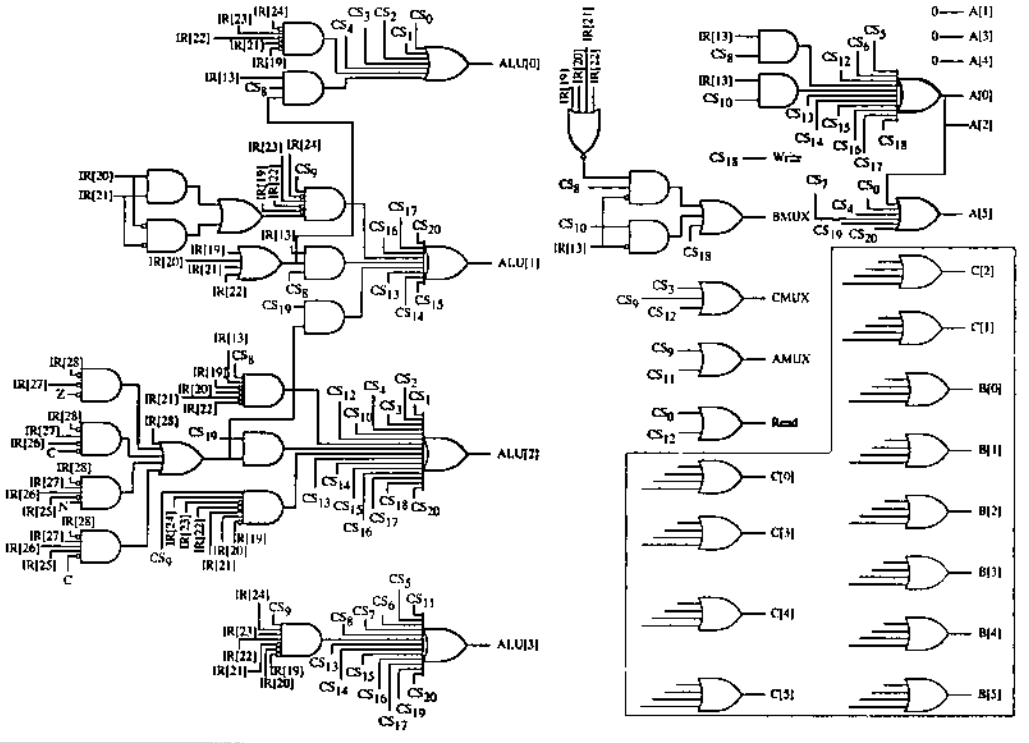


Figura 6.24 • La sección de control cableada en ARC: señales que van desde la sección de datos de la unidad de control al trayecto de datos. (No se detallan las áreas sombreadas.)

La señal de lectura (*Read*) se coloca en 1 en las líneas 0 y 12, por lo que se la obtiene a partir de la suma lógica de CS_0 y CS_{12}. La señal de escritura (*Write*) solamente se genera en la línea 18, por lo que no necesita lógica alguna, más que la línea CS_{18}.

El control de la unidad aritmético-lógica se realiza a través de cuatro señales: ALU[0], ALU[1], ALU[2] y ALU[3], las que corresponden a F_0, F_1, F_2 y F_3 de la tabla de operación de la unidad aritmético-lógica de la figura 6.4. Estas cuatro señales requieren valores en cada una de las 20 líneas del programa HDL. En la línea 0, la operación de la unidad aritmético-lógica es un producto lógico Y, que corresponde a ALU[3:0] = 0101. La línea 1 no especifica operación aritmético-lógica alguna, por lo que en forma arbitraria se elige una que no produzca efectos secundarios, como la operación AND (0101). Si se sigue por el mismo camino y se toman en cuenta las sentencias condicionadas (`CONDITIONED ON`), se obtiene la lógica correspondiente a las señales ALU[3:0], que se indica en la figura.

Las señales de control se envían al trayecto de datos, en forma similar al control que realiza el MIR sobre el trayecto de datos en la definición microprogramada de la figura 6.10. Las soluciones cableada y microprogramada pueden considerarse como intercambiables, excepto por los costos involucrados. En la solución cableada solo se tienen 21 *flip flops*, contra 2.048 x 41 = 83.968 *flip flops* en la solución microprogramada (si bien es cierto que el uso de una memoria de lectura ocuparía menos espacio por permitir la uti-

lización de elementos de almacenamiento más pequeños que un *flip flop*). La cantidad de lógica combinatoria adicional es comparable. La solución cableada es más veloz con respecto a la ejecución de las instrucciones de ARC, en especial en la decodificación de las instrucciones con formato de salto, pero se hace más difícil su modificación una vez iniciada su fabricación.

Ejemplo

Considérese el agregado de la misma instrucción `subcc` del ejemplo anterior a la implementación cableada del conjunto de instrucciones de ARC. Tal como en el caso anterior, la instrucción subcc utiliza el formato aritmético y un campo op3 de 001100.

Solo se requiere la modificación de la línea 9 del código HDL, en la que se inserta la expresión:

```
ADDCC (R[rs1], INC_1 (temp0)) CONDITIONED ON XNOR (IR[19:24], 001100, ! subcc
```

antes de la línea correspondiente al `addcc`.

Las señales que requieren modificación son `ALU[3:0]`. La construcción `INC_1` de la línea precedente indica que debería crearse un sumador combinatorio, que podría estar definido en algún otro módulo HDL. (En una unidad de control cableada, existe gran flexibilidad en cuanto a lo que se puede hacer.) ∘

6.4 Estudio de un caso: el lenguaje de descripción de hardware VHDL

Esta sección presenta una breve descripción de *VHDL (VHSIC Hardware Description Language*, en la que VHSIC es el acrónimo de *Very High Speed Integrated Circuit*; literalmente, lenguaje de descripción de hardware para circuitos integrados de muy alta velocidad). Los lenguajes descriptivos de hardware (HDL), como VHDL y AHPL, son lenguajes que se utilizan para la descripción de los circuitos de computadoras, enfocados principalmente al diseño de los dispositivos lógicos y de los circuitos integrados. En el caso de VHDL, no obstante, pueden especificarse los diseños en muchos niveles diferentes. Por ejemplo, la unidad de control implementada en este capítulo podría especificarse en VHDL.

Se analizan primero los fundamentos que llevaron al desarrollo de VHDL, tras lo cual se describen algunas de sus propiedades. Luego, se analiza una especificación VHDL de la función mayoría.

6.4.1 Antecedentes

VHDL surgió como resultado de la colaboración entre el Departamento de Defensa de los Estados Unidos y muchas industrias estadounidenses. El Departamento de Defensa, principalmente a través de su Agencia de Desarrollo Avanzado (DARPA, *Defense Advanced Research Products Agency*), descubrió, a fines de los años 1970, que el diseño y fabricación de circuitos integrados se estaba volviendo tan complejo que se hacia necesaria la creación de un conjunto de herramientas de diseño integradas, tanto para el diseño como para la simulación. Se hacía sentir que las herramientas debían permitir que el usuario pudiese especificar un circuito o sistema a partir de su nivel más alto, el de su comportamiento, bajando desde ahí a los menores niveles correspondientes al diseño y configuración de los circuitos integrados reales. Más aún, todas estas especificaciones debían ser pasibles de verificación por medio de simuladores y otros tipos de elementos de verificación de reglas.

La primera definición de requerimientos del lenguaje fue generada por el Departamento de Defensa en 1981, como reconocimiento de la necesidad de un enfoque más consistente en el diseño de hardware de computadoras. El contrato para el desarrollo de la primera versión del lenguaje fue adjudicado a un consorcio formado por IBM, Texas Instruments e Intermetrics, una empresa de ingeniería de software especializada en el diseño e implementación de lenguajes de programación.

El consorcio entregó en 1985 una versión preliminar para ser probada y comentada. Una versión más actualizada se envió al IEEE en 1986 para su normalización, obteniéndose como resultado la norma IEEE 1076-1987. En 1993 se aprobó una nueva versión, IEEE 1076-1993, para resolver algunos problemas menores y agregar algunas prestaciones nuevas.

Se mida por donde se mida, VHDL es un éxito, con gran cantidad de usuarios tanto dentro como fuera de la comunidad de contratistas del Departamento de Defensa. Los actuales requerimientos del Departamento de Defensa implican que cualquier circuito integrado de aplicación específica (ASIC, *Application Specific Integrated Circuit*) debe venir acompañado de su modelo VHDL con propósitos de prueba y simulación. Casi todos los fabricantes de herramientas de diseño asistido por computadora incluyen en sus sistemas soporte para VHDL.

6.4.2 ¿Qué es VHDL?

En los términos más básicos, VHDL es un lenguaje descriptivo de hardware que puede utilizarse para la descripción y modelización de sistemas digitales. VHDL tiene implícito el sentido del tiempo y puede administrar la progresión de eventos a lo largo del tiempo. A diferencia de la mayoría de los lenguajes de procedimiento de uso común, VHDL soporta **ejecución concurrente** y es **controlado por eventos.**

Ejecución concurrente

El concepto de ejecución concurrente significa que, a menos que se realicen esfuerzos especiales para especificar una ejecución secuencial, todas las sentencias de una especificación VHDL se ejecutan en paralelo. Esta es la forma en que debiera ocurrir, dado que cuando se alimenta eléctricamente un sistema digital, el sistema funciona "en paralelo". Esto es, la corriente eléctrica circula por los circuitos de acuerdo con las leyes de la física y de la lógica, sin ningún sentido implícito referido a "quien llegó primero".

Sistema manejado por eventos

VHDL opera con señales que se propagan a través de sistemas digitales y, por lo tanto, debe soportar lógica y naturalmente el concepto de los cambios de estado como función del tiempo. Porque tiene sentido del tiempo, soporta conceptos como "luego de", "hasta que" y "espera". Como sistema controlado por eventos, inicia su funcionamiento por medio de la ejecución de cualquier código de inicialización, y luego registra todos los cambios en los valores de las señales, de 0 a 1 y de 1 a 0, que ocurran en las entradas y salidas de sus componentes. Registra estos cambios, o eventos, en una cola ordenada cronológicamente, conocida como cola de eventos. Examina estos eventos, y si un evento dado produce algún efecto sobre algún componente, evalúa dicho efecto. Si el efecto produce la ocurrencia de nuevos efectos posteriores, el simulador coloca estos nuevos eventos en la cola, y el proceso continúa hasta, y a menos, que no haya otros nuevos eventos para procesar.

Niveles de abstracción y descomposición jerárquica

Como se ha mencionado, las especificaciones para VHDL pueden escribirse en casi cualquier nivel de abstracción, desde el nivel puramente algorítmico, en el que se especifica el comportamiento a partir de algoritmos formales, hasta el nivel lógico, en el que la conducta se especifica a través de expresiones booleanas.

Más aún, una especificación VHDL puede estar constituida por una jerarquía de componentes, lo que significa que un componente puede contener componentes, los que a su vez también pueden contener componentes. Esto representa un modelo del mundo físico, en el que, por ejemplo, una placa de circuito impreso puede contener circuitos integrados, los que a su vez están compuestos por módulos, que también contienen submódulos, hasta llegar al nivel inferior de las compuertas lógicas individuales y, más abajo aún, al de los transistores.

6.4.3 La función mayoría y su descripción en lenguaje VHDL

Se analizará la forma en la que puede utilizarse VHDL para implementar un componente digital sencillo. Esto se realizará a través de varias implementaciones de la **función mayoría**, la que produce un 1 en la salida si más de la mitad de sus entradas valen 1, entre-

gando un 0 en caso contrario. Es una función útil en el análisis de tolerancia de fallas, en los que múltiples sistemas realizan la misma operación sobre el mismo conjunto de datos y "votan", por lo que si uno de los sistemas se desvía de los demás, su salida se ignora. La función mayoría se analiza en forma detallada en el apéndice A. Su tabla de verdad se muestra en las figuras A.15 y A.16 y se reproduce en este capítulo como figura 6.25.

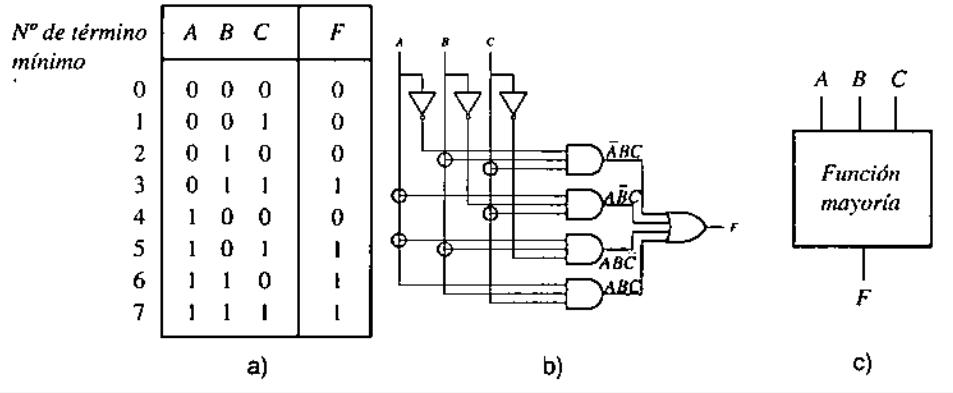


Figura 6.25 • La función mayoría: (a) tabla de verdad, (b) implementación Y-O, (c) representación como caja negra.

En VHDL, la especificación de cualquier componente, como la función mayoría, se divide en dos partes, una que define la **entidad** y otra que corresponde a la **arquitectura**. Estas dos partes concuerdan, de alguna manera, con las partes sintáctica y semántica de la especificación de un lenguaje. La entidad describe la interfaz del componente sin mencionar nada acerca de su estructura interna. La arquitectura describe su comportamiento interno. A continuación se presenta una especificación de entidad de la función mayoría implementada con tres entradas:

```
- - Interfaz
entity MAJORITY is
port
    (A_IN, B_IN, C_IN : in BIT
    F_OUT             : out BIT);
end MAJORITY
```

Las palabras clave se destacan en negrita y los comentarios se inician con "--" y se terminan al final del renglón. Las distintas sentencias se separan con punto y coma.

La especificación de **entity** describe simplemente las señales de entrada y salida a la caja negra de la figura 6.25c. La declaración **port** describe el tipo de entradas que ingresan y salen de la entidad. Los modos de port incluyen **in** para señales que ingresan a la entidad, **out** para las que salen de la misma, e **inout** para señales bidireccionales. Para port existen también otros modos que están orientados a aplicaciones especiales.

Habiendo especificado la interfaz al componente, se puede plantear ahora el funcionamiento interno del mismo utilizando la especificación de **arquitectura**:

Modelo conductista correspondiente a la función mayoría

```
-- Cuerpo
architecture LOGIG_SPEC of MAJORITY is
begin
-- Calcular la salida usando una expresiòn booleana
F_OUT      <= (not A_IN and B_IN and C_IN) or
               (A_IN and not B_IN and C_IN) or
               (A_IN and B_IN and not C_IN) or
               (A_IN and B_IN and C_IN) after 4 ns;
end LOGIC_SPEC
```

Este modelo describe la relación entre la declaración de la entidad de MAJORITY (la función mayoría) y su arquitectura. Los nombres A_IN, B_IN, C_IN y F_OUT del modelo de la arquitectura deben coincidir con los nombres utilizados en la declaración de la entidad.

Este tipo de especificación arquitectónica se denomina conductista, debido a que define la función de entrada-salida en base al enunciado de una función explícita de transferencia. Esta función es una expresión booleana que implementa la función booleana de las figuras 6.25a y b. Debe notarse, no obstante, que aun en este nivel de especificación se puede incorporar un retardo de tiempo entre entradas y salidas por medio de la palabra clave **after**. En este caso, el evento que genera el valor de F_OUT se disparará 4 ns después de un cambio en cualquiera de los valores de entrada.

También es posible especificar la arquitectura en un nivel más cercano al hardware por medio de la especificación de compuertas lógicas en lugar de ecuaciones lógicas. En este caso se habla de un modelo estructural. El siguiente es un ejemplo de tal especificación.

Modelo estructural del componente mayoría

En la generación del modelo estructural de la entidad MAJORITY convendrá seguir el diseño de compuertas especificado en la figura 6.25b. El modelo se inicia con la descripción de un conjunto de operadores lógicos, en una construcción especial de VHDL conocida como un **package** (paquete). El paquete se supone almacenado en una biblioteca de trabajo llamada WORK. A continuación de la especificación del paquete se repite la declaración de **entity**, y luego, con las declaraciones del paquete y de la entidad, se especifica el funcionamiento interno del componente por medio de la especificación de la arquitectura a nivel estructural:

```
-- Declaración del paquete en la biblioteca WORK
package LOGIC_GATES is
component AND3
   port (A, B, C : in BIT; X : out BIT);
end component;
component OR4
   port (A, B, C, D : in BIT; X : out BIT);
end component;
component NOT1
   port (A : in BIT; X : out BIT);
end component;

-- Interfaz
entity MAJORITY is
   port
        (A_IN, B_IN, C_IN    : in BIT
        F_OUT                : out BIT);
end MAJORITY

-- Cuerpo
-- Usa los componentes declarados en el paquete LOGIC_GATES
-- en la biblioteca WORK
-- Importar todos los componentes de WORK.LOGIC_GATES
use WORK.LOGIC_GATES .all
architecture LOGIC_SPEC of MAJORITY IS
-- Declarar las señales utilizadas internamente en la función
   MAJORITY
signal A_BAR, B_BAR, C_BAR, I1, I2, I3, I4: BIT;
begin
-- Conectar las compuertas lógicas
NOT_1: NOT1 port map (A_IN, A_BAR);
NOT_2: NOT1 port map (B_IN, B_BAR);
NOT_3: NOT1 port map (C_IN, C_BAR);
AND_1: AND3 port map (A_IN, B_IN, C_IN, I1);
AND_2: AND3 port map (A_IN, B_BAR, C_IN, I2);
AND_3: AND3 port map (A_IN, B_IN, C_BAR, I3);
AND_4: AND3 port map (A_IN, B_IN, C_IN, I4);
OR_1: OR4 port map (I1, I2, I3, I4, F_OUT);
end LOGIC_SPEC;
```

La declaración **package** provee tres compuertas, una compuerta Y de tres entradas, AND3, una compuerta O de cuatro entradas, OR4, y una compuerta inversora, NOT1. Las arquitecturas de estos elementos se suponen declaradas en alguna parte del paquete. La declaración **entity** se mantiene inalterada, como se podría suponer, dado que especifica MAJORITY como una caja negra.

El cuerpo de la especificación se inicia con una cláusula **use**, que importa todas las declaraciones del paquete LOGIC_GATES incluidas en la biblioteca WORK. La decla-

ración **signal** define siete señales binarias BIT que se utilizarán internamente. Estas señales se usan para interconectar los componentes dentro de la arquitectura.

A continuación se declaran las tres compuertas inversoras, NOT_1, NOT_2 y NOT_3, las cuales se corresponden con el modelo NOT1 de compuerta inversora. Se especifican sus señales de entrada y salida luego de las palabras clave **port map**. Las señales a la entrada y salida de las compuertas lógicas se configuran de acuerdo con el orden en que fueron declaradas dentro del paquete.

El resto del cuerpo de la especificación conecta las compuertas NO, las compuertas Y y la compuerta O, tal como se muestra en la figura 6.25b.

Nótese que esta forma de especificación de la arquitectura separa el diseño y la implementación de las compuertas lógicas del diseño de la entidad MAJORITY. Sería posible tener diferentes implementaciones de las compuertas lógicas en diferentes paquetes y utilizar cualquiera de ellas simplemente reemplazando la cláusula **uses**.

6.4.4 Sistema lógico de nueve valores

Este tratamiento breve de VHDL solo ofrece una pequeña imagen del alcance y del poder del lenguaje. El lenguaje completo contiene herramientas para especificar señales de reloj y distintos mecanismos de temporización, procesos secuenciales y diferentes tipos de señales. Existe una norma IEEE que define un sistema lógico de nueve elementos, conocida como STD_ULOGIC, IEEE 1164-1993. La misma contiene los siguientes valores lógicos:

```
type STD_ULOGIC is (
'U', — No inicializado
'X', — Forzando desconocido
'0', — Forzando 0
'1', — Forzando 1
'Z', — Alta impedancia
'W', — Desconocido débil
'L', — Cero débil
'H', — Uno débil
'-', — Irrelevante
);
```

Sin entrar en demasiado detalle, estos valores le permiten al usuario detectar defectos lógicos dentro de un diseño y rastrear en él la propagación de señales no inicializadas o débiles.

Resumen

Una microarquitectura consiste en una sección de control y un trayecto de datos. El trayecto de datos contiene registros de datos, una unidad aritmético-lógica y las conexiones entre dichos elementos. La sección de control contiene registros para las microinstrucciones (para el caso de una solución microprogramada) y para los códigos de condición, así como un controlador. El controlador puede ser microprogramado o cableado. Un controlador microprogramado interpreta microinstrucciones por medio de la ejecución de un microprograma almacenado en una memoria de control. Una unidad de control cableada está organizada como un conjunto de *flip flops*, que mantienen información de estado, y lógica combinatoria, que implementa las transiciones entre los estados.

La solución cableada es veloz y requiere poca electrónica en comparación con la alternativa microprogramada. Esta última es flexible y simplifica el proceso de modificación del conjunto de instrucciones. La memoria de control consume una gran porción de hardware, la que puede reducirse en un diseño que tome en cuenta el uso de la nanoprogramación. La nanoprogramación agrega retardo al ciclo de ejecución de las microinstrucciones. Decidir entre una solución microprogramada y una cableada lleva a soluciones de compromiso. La solución microprogramada es grande y lenta pero es flexible y permite una implementación simple, en tanto que la solución cableada es rápida y pequeña pero difícil de modificar, dando como resultado implementaciones más complejas.

Para lectura posterior

M. V. Wilkes y otros es un clásico referido a microprogramación. J. Mudge cubre el tema de la microprogramación en las computadoras DEC PDP 11/60. A. Tanenbaum y M. Mano proveen ejemplos instructivos de arquitecturas microprogramadas. F. J. Hill y G. R. Peterson presentan un tratamiento tutorial del lenguaje descriptor de hardware AHPL y del control cableado en general. R. Lipsett y otros, y Z. Navabi describen el lenguaje comercial VHDL y ofrecen ejemplos de su utilización. D. Gajski cubre varios aspectos de la compilación de silicio.

Gajski, D., *Silicon Compilation*, Addison Wesley, 1988.

Hill, F. J. y G. R. Peterson, *Digital Systems: Hardware Organization and Design*, 3ª ed., John Wiley & Sons, 1987.

Lipsett, R., C. Schaefer y C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, 1989.

Mano, M., *Digital Design*, 2ª ed., Prentice Hall, 1991. (Traducción al español disponible: *Diseño digital*, Prentice Hall, 1987.)

Mudge, J. Craig, "Design Decisions for the PDP 11/60 Mid Range Minicomputer", en: *Computer Engineering, A DEC View of Hardware Systems Design*, Digital Press, 1978.

Navabi, Z., *VHDL: Analisys and Modeling of Digital Systems*, McGraw-Hill, 1993.

Tanenbaum, A., *Structured Computer Organization*, 3ª ed., Prentice Hall, 1990. (Traducción al español disponible: *Organización de computadoras, un diseño estructurado*, 4ª ed., Prentice Hall, 2000.)

Wilkes, M. V., W. Redwick y D. Wheeler, "The Design of a Control Unit of an Electronic Digital Computer", en: *Proc. IRE*, vol. 105, 1958, p. 21.

Problemas

6.1 Diseñar una unidad aritmético-lógica de un bit, utilizando el circuito de la figura 6.26, el que permite realizar las operaciones de suma aritmética, Y, O y NO sobre las entradas *A* y *B*, de un bit cada una. En cada operación se genera una salida *Z* de un bit, en tanto que la operación de suma también genera un arrastre. El arrastre será 0 para las operaciones Y, O y NO. Diseñar la unidad aritmético-lógica de un bit utilizando los componentes que aparecen en el diagrama. Indicar las conexiones entre los componentes. No agregar compuertas, multiplexores ni elemento adicional alguno. *Nota:* El sumador completo recibe dos entradas de un bit (*X* e *Y*) y un arrastre de entrada, y produce una suma y un arrastre de salida.

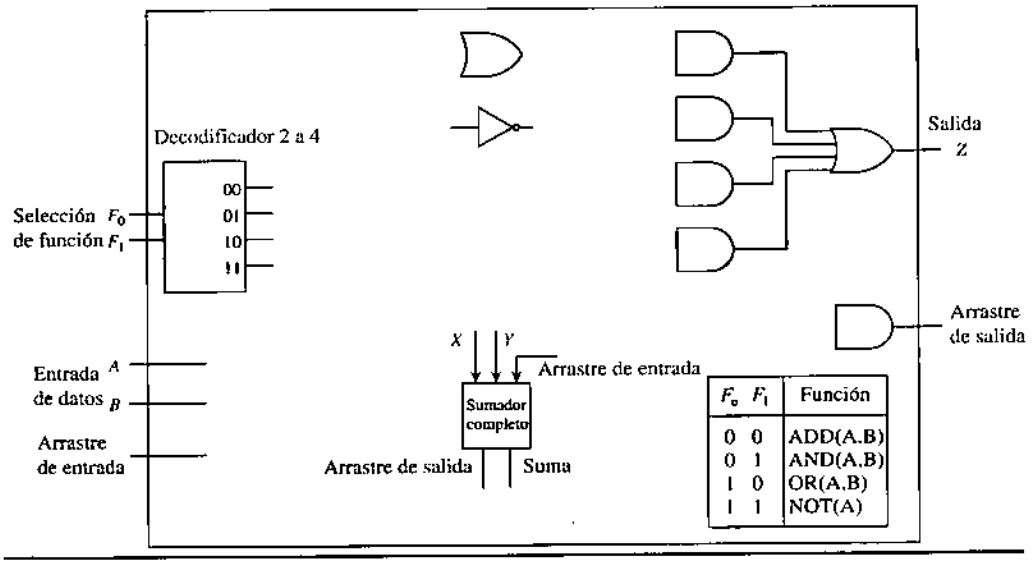


Figura 6.26 • Una unidad aritmético-lógica de 1 bit.

6.2 Diseñar una unidad aritmético-lógica que reciba dos operandos de 8 bits *X* e *Y* y que produzca una salida *Z* de 8 bits. Existe una entrada de control C de dos bits en la que 00 selecciona la operación Y, 01 selecciona la operación O, 10 selecciona NOR y 11 selecciona XOR. Al diseñar la unidad aritmético-lógica deberán seguirse estos procedimientos: (1) dibujar un diagrama en bloques de ocho unidades aritmético-lógicas de un bit, cada una de las cuales aceptará uno de los bits de *X* y de *Y*, y los dos bits de control, produciendo la correspondiente salida *Z* de un bit; (2) crear una tabla de verdad que describa la unidad aritmético-lógica de un bit; (3) diseñar una de las unidades aritméticas de un bit utilizando un multiplexor de 8 entradas de datos.

6.3 Diseñar una unidad de control para un videojuego sencillo en el que aparece un carácter en la pantalla que caza objetos. Considerar el problema como una máquina de estados finitos, de la que solo se pide representar el diagrama de transiciones de estados. No se pide desarrollar el circuito de la máquina. La entrada a la unidad de control es un vector de dos bits, en el que 00 indica mover a izquierda, 01 indica mover a derecha, 10 indica no mover y 11 indica detenerse. La salida Z es 11 si la máquina está detenida, y es 00, 01 o 10, de acuerdo con los patrones de entrada. Una vez que la máquina se detiene, permanece en ese estado indefinidamente.

6.4 En la figura 6.3 no hay conexión desde la salida del decodificador C al registro `%r0`. Justificar el motivo.

6.5 Con referencia a la figura 6.27, los registros 0, 1 y 2 son registros de uso general. El registro 3 se inicializa al valor +1, el que puede ser modificado por el microcódigo. Se requiere la certeza de que el mismo no se modifica.

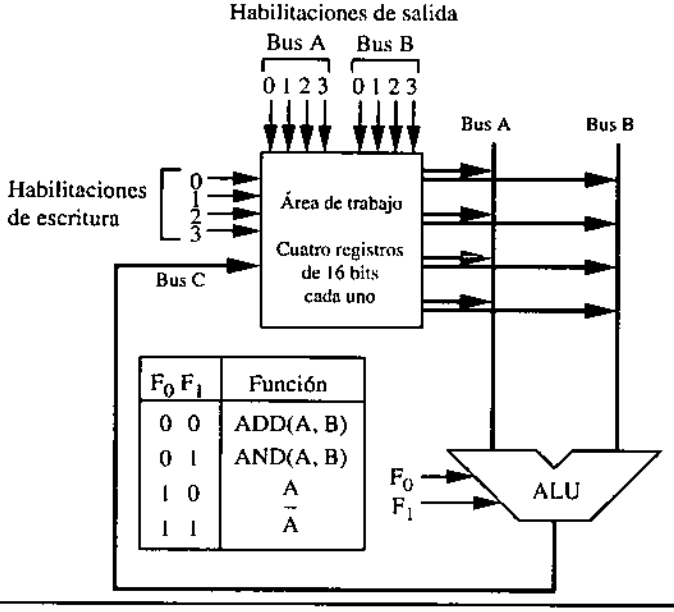


Figura 6.27 • Una microarquitectura pequeña.

a. Escribir una secuencia de control que calcule la diferencia entre los contenidos de los registros 0 y 1, en representación de complemento a 2, y deje el resultado en el registro 0. Simbólicamente, se puede expresar lo pedido como r0 ← r0-r1. No modificar registro alguno excepto r0 y r1 (si hiciera falta). Completar la tabla siguiente con ceros o unos (usando ceros cuando el valor a asignar sea irrelevante). Suponer que si no se selecciona registro alguno para los buses A o B, el bus adopta el valor 0.

Habilitaciones de escritura				Habilitaciones del bus A				Habilitaciones del bus B						
0	1	2	3	0	1	2	3	0	1	2	3	F_0	F_1	Tiempo
														0
														1
														2

b. Escribir una secuencia de control que calcula la XOR de los contenidos de los registros 0 y 1, dejando el resultado en el registro 0. Simbólicamente, se puede expresar lo pedido como r0 ←XOR(r0, r1). Utilizar el mismo estilo de solución que para la parte a.

6.6 Escribir el formato binario de las microinstrucciones que se indican. Usar el estilo de la figura 6.17. Utilizar 0 como valor a asignar a cualquier campo que no sea necesario.

```
60: R [temp0] ← NOR(R[0],R[temp0]); IF Z THEN GOTO 64;
61: R[rd] ← INC(R[rs1]);
```

6.7 Se muestran tres palabras binarias, cada una de las cuales puede interpretarse como una microinstrucción. Escribir la versión simbólica de cada una de las palabras binarias utilizando el lenguaje microensamblador desarrollado a lo largo del capítulo.

	A MUX		B MUX		C MUX					
A	X	B	X	C	XDR	ALU	COND	JUMP ADDR		
100101	0	000000	0	100001	0	0	0	1100	000	000000000000
000000	1	100001	0	100001	0	0	0	1000	110	111000000001
000000	1	000000	1	100001	0	0	0	1000	101	111000100100

6.8 Escribir de nuevo el microcódigo de la instrucción `call` que comienza en la línea 1280 de modo de utilizar solo tres líneas de microcódigo en lugar de cuatro. Usar la operación LSHIFT2 una única vez en vez de usar dos veces ADD.

6.9 a. ¿Cuántas instrucciones se ejecutan para interpretar la instrucción `subcc` que fuera agregada en el primer ejemplo del capítulo? Escribir los números de microinstrucciones en el orden en que se ejecutan, comenzando con la microinstrucción 0.

b. Usando la alternativa cableada para representar el microcontrolador ARC, ¿cuántos estados requiere la ejecución de la instrucción `addcc`? Indicar los estados en el orden en que se ejecutan, empezando desde el estado 0.

6.10 a. Listar las microinstrucciones que se ejecutan al interpretar la instrucción BA.

b. Listar los estados (figura 6.22) que se recorren al interpretar la instrucción BA.

6.11 El registro `%r0` puede diseñarse usando solo circuitos *buffer* de tres estados. Representar este diseño.

6.12 ¿Qué formato binario debe colocarse en el campo C de una micropalabra si no se pretende cambiar el contenido de registro alguno?

6.13 La figura 6.28 ilustra una unidad de control para una máquina herramienta. Se requiere crear el microcódigo para la misma. La máquina se comporta de la siguiente forma: si la entrada de detención *A* se coloca alguna vez en 1, la salida de la máquina queda detenida para siempre y se genera un 1 perpetuo en la salida *X*, a la vez que se genera un 0 en las salidas *V* y *W*. Cuando no hay ninguna entrada habilitada se habilita (se coloca en 1) una luz de espera (salida *V*). Esto significa que V se enciende cuando las entradas *A*, *B* y *C* valen 0 y la máquina no está detenida. Con cada evento que se produce en las entradas (*B* = 1 y/o *C* = 1) se hace sonar una campanilla (*W* = 1), excepto cuando la máquina está detenida. La entrada *D* y la salida *S* pueden utilizarse para información de estado en el microcódigo a desarrollar. Cualquier campo que no se utilice puede adoptar el valor 0. *Sugerencia*: Llenar primero la mitad inferior de la tabla.

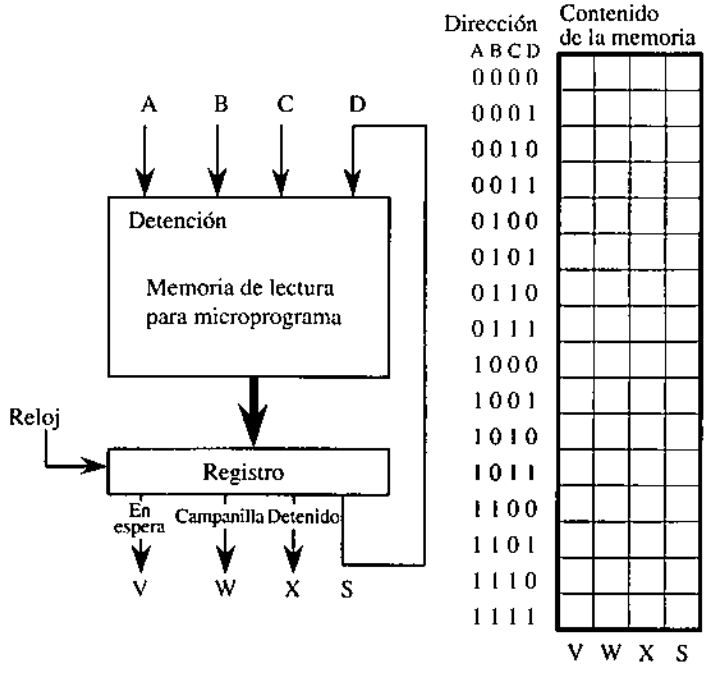
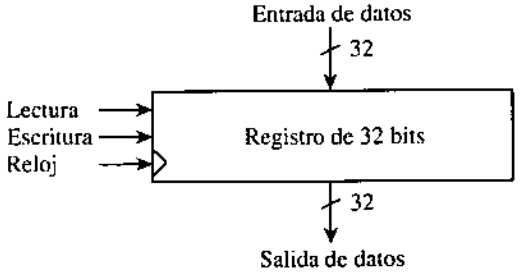


Figura 6.28 • Unidad de control para una máquina herramienta.

6.14 Se requiere extender el conjunto de instrucciones de ARC con el objeto de incluir una nueva instrucción a través de la modificación del microprograma. La instrucción a ser codificada es `xorcc`, lo que implica calcular la XOR de los operandos y fijar apropiadamente los códigos de condición. Esta operación debe adoptar el formato aritmético; el campo `op3` es 010011.

Indicar las nuevas microinstrucciones a incorporar para poder ejecutar `xorcc`.

6.15 Plantear el diseño de una pila de registros de cuatro palabras, usando registros de 32 bits que tienen la siguiente estructura:



Se apilan cuatro registros de modo que la salida del registro superior es la entrada al segundo registro, el que a su vez entrega sus salidas en la entrada del tercer registro, el cual, a su vez, genera con su salida la entrada al cuarto registro. La entrada a la pila se asigna al registro superior, en tanto que la salida de la pila se toma desde la salida del registro superior (no del inferior). Existen dos líneas de control adicionales, `push` y `pop`, que provocan el ingreso de los datos a la pila y la extracción de los datos desde la pila, respectivamente, cuando la línea correspondiente vale 1. Si ninguna de las líneas es 1, o si ambas son 1 en forma simultánea, no se altera el contenido de la pila.

6.16 En la línea 1792 del microprograma de ARC, aparece un `GOTO` condicional al final de la línea, en tanto que en la línea 8 aparece al principio de la misma. ¿Interesa la posición de una sentencia de `GOTO` dentro de una línea de microcódigo?

6.17 La figura 6.29 ilustra una microarquitectura particular. El trayecto de datos tiene cuatro registros y una unidad aritmético-lógica. La sección de control es una máquina de estados finitos, en la que existe una memoria de acceso aleatorio (RAM) y un registro. En esta microarquitectura, un programa compilador traduce directamente un programa de alto nivel hacia el microcódigo; no hay un formato simbólico ensamblador intermedio, por lo que tampoco hay ciclos de búsqueda ni de ejecución.

Se requiere escribir el microcódigo que implementa las instrucciones indicadas al pie del problema. El microcódigo debería estar almacenado en las posiciones 0, 1, 2 y 3 de la memoria de acceso aleatorio. Si bien no hay líneas que así lo indiquen, deberá interpretarse que los bits n y z son ambos 0 cuando C_0C_1 son 00. Esto significa que si no hay posibilidad de salto, las líneas A_{22} y A_{23} son ambas 0. *Nota*: Cada bit de los campos A, B y C corresponde directamente a un registro. Así, la palabra 1000 selecciona el registro R3, no el registro 8, inexistente. Existen algunas complicaciones en lo que hace a la resolución de los saltos en esta microarquitectura. Las mismas no deberían afectar a la generación del microcódigo.

```
0: R1 ← ADD (R2, R3)
1: Saltar por negativo a (15)10
2: R3 ← AND (R1, R2)
3: Saltar a (20)10
```

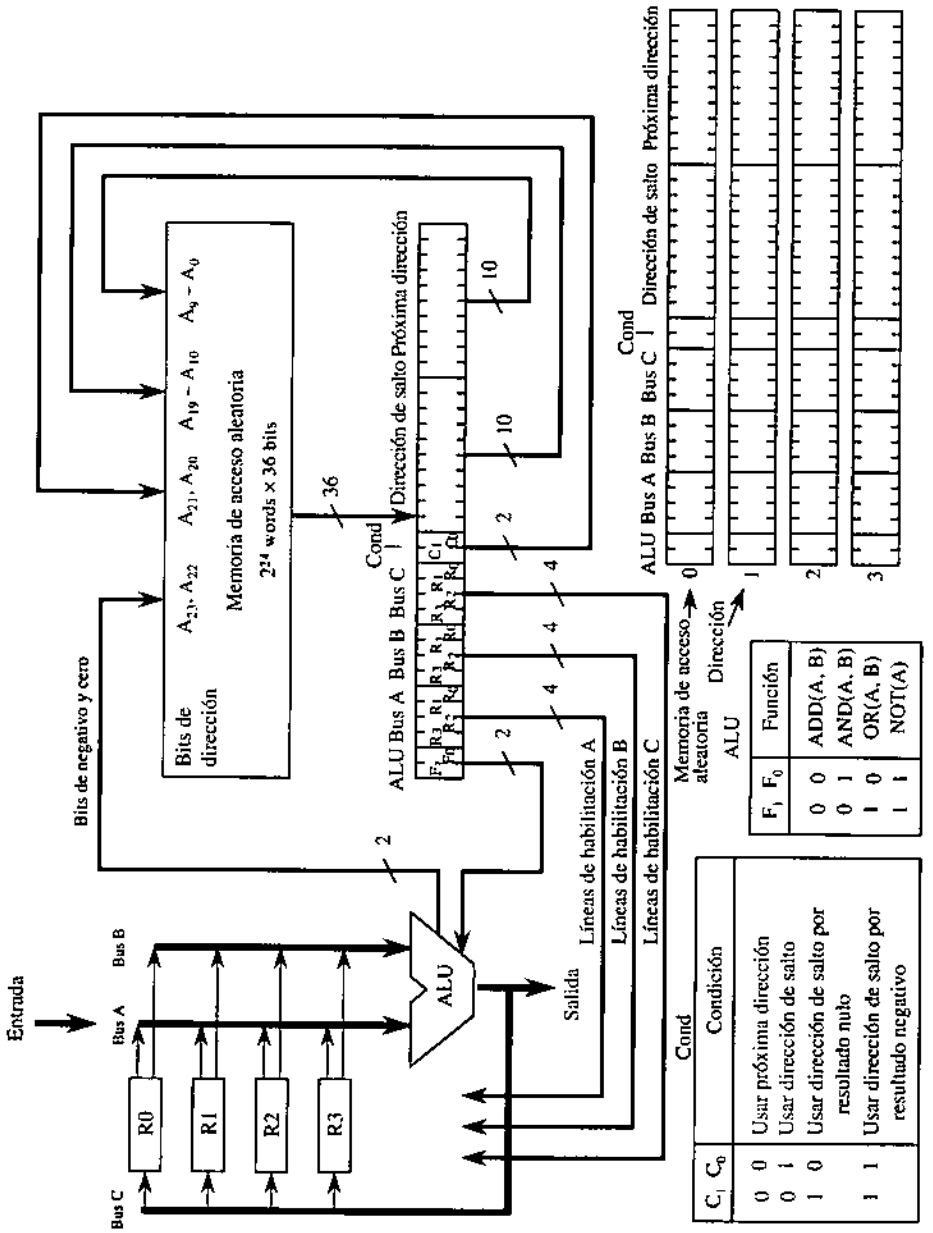


Figura 6.29 • Un ejemplo de microarquitectura.

6.18 En la línea 2047 del microprograma ARC que se muestra en la figura 6.15, ¿se comportaría en forma diferente el programa si se eliminara la parte de la instrucción que incluye el GOTO 0?

6.19 En la **microprogramación horizontal** las micropalabras son anchas (de muchos bits), en tanto que en **la microprogramación vertical** las palabras son angostas. En general, las palabras de microcódigo horizontal se pueden ejecutar rápidamente pero requieren mucho más espacio que las micropalabras verticales, las que llevan más tiempo de ejecución. Si se convierte el formato de microcódigo de la figura 6.11 en un formato más horizontal, por medio de la expansión de los campos A, B y C para que contengan un solo bit para cada uno de los registros en lugar de una versión codificada en 6 bits, se podrán eliminar los decodificadores A, B y C de la figura 6.3. Esto permite incrementar la frecuencia de funcionamiento pero aumenta también el tamaño de la memoria de microprograma.

a. ¿Qué ancho tendrán las nuevas palabras del microcódigo horizontal?

b. ¿Cuál es el porcentaje de incremento de tamaño del microcódigo?

6.20 Con referencia a la figura 6.7, indicar las entradas a las tablas de busqueda LUT_0 y LUT_x (con $x > 0$) de la unidad aritmético-lógica para una instrucción INC(A).

6.21 En algunas arquitecturas existe una parte especial del hardware que actualiza el contador de programa, tomando en cuenta que los dos bits menos significativos son siempre 0. En este capítulo no se ha planteado la existencia de circuito alguno para actualizar el contador de programa, y el microcódigo incluido en las líneas 2–20 de la figura 6.15, correspondiente a las instrucciones de salto, presenta un error en cuanto a la forma en que se actualiza el contador de programa (específicamente en la línea 12) debido a que los desplazamientos de los saltos se expresan en términos de palabras. Identificar el error e indicar la forma de repararlo.

Capítulo 7

Memoria

En las últimas décadas, la velocidad de procesamiento de las unidades de proceso, medidas por la cantidad de instrucciones ejecutadas en un segundo, se ha duplicado cada 18 meses sin variar su precio. La memoria de las computadoras ha experimentado un incremento similar, pero considerando un parámetro diferente, cuadruplicando su tamaño cada 36 meses por el mismo precio. No obstante, la velocidad de las memorias ha ido aumentando a razón de menos de un 10% anual. Así, mientras la velocidad de procesamiento aumenta en la misma proporción en que aumenta el tamaño de las memorias, también aumenta la brecha entre la velocidad del procesador y la velocidad de las memorias.

A medida que aumenta la brecha entre las velocidades de procesador y de memoria, las soluciones arquitectónicas buscan tender un puente sobre esa brecha. Una computadora típica suele contener distintos tipos de memorias, que van desde la memoria cara y rápida de los registros internos (véase el apéndice A) hasta las memorias baratas y lentas de los discos removibles. La interacción entre estos diferentes tipos de memoria se aprovecha de forma tal que se logra un comportamiento, por parte de la computadora, equivalente al que tendría si tuviera una memoria única, grande y rápida, cuando en realidad contiene una variedad de tipos de memorias que operan de un modo altamente coordinado. Este capítulo comienza con un análisis de alto nivel sobre la forma en que se organizan estos distintos tipos de memorias en lo que se conoce como **jerarquía de memorias**.

7.1 Las jerarquías de la memoria

La memoria de una computadora digital convencional se encuentra organizada bajo un criterio jerárquico, como el que ilustra la figura 7.1. En la cima de la jerarquía se encuentran los registros, de velocidad similar a la de la unidad de proceso, pero grandes y consumidores de una importante cantidad de energía de alimentación. En un procesador se encuentran habitualmente unos pocos registros, del orden de algunos cientos o menos. Al fondo de la jerarquía aparecen las memorias secundarias y los elementos de almacenamiento "*off line*", tal como los discos magnéticos rígidos y las cintas magnéticas, en

los que el costo por bit almacenado es bajo en términos monetarios y de energía consumida, pero cuyo tiempo de acceso es muy alto comparado con el de los registros. Entre los registros y los elementos de almacenamiento secundario se ubican otros tipos de memorias que tienden a salvar la brecha entre ambos extremos.

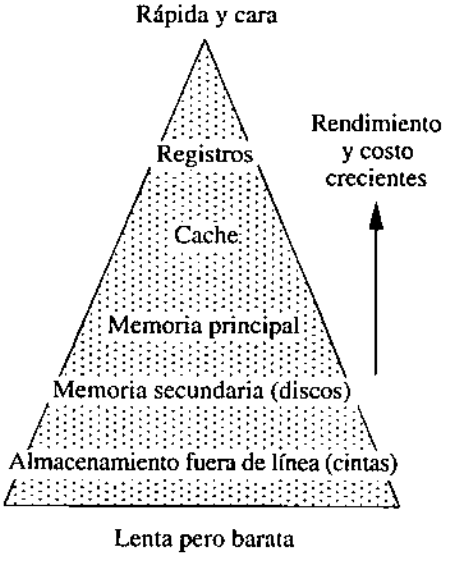


Figura 7.1 • La jerarquía de las memorias.

A medida que se recorre la estructura jerárquica, se obtiene una mayor eficiencia a cambio de un mayor costo. La tabla 7.1 ilustra algunas de las propiedades de los componentes de las diferentes jerarquías de memoria, consideradas a fines del siglo XX (finales de los años noventa). Nótese que el costo típico, al que se arriba multiplicando el costo por Mbyte por la cantidad de memoria utilizada en una máquina, es similar para cada uno de los escalones de la jerarquía. Nótese asimismo que el tiempo de acceso varía en factores aproximadamente de 10, excepto para el caso de los discos, cuyos tiempos de acceso son del orden de 100.000 veces mayores que los de la memoria principal. Este gran desajuste influye fuertemente sobre la forma en que el sistema operativo debe manejar la transferencia de bloques de datos entre discos y memoria principal, como se verá a lo largo del capítulo.*

* *N. de T.*: Debido a las consideraciones planteadas al comienzo del capítulo por los autores, al momento de la publicación de la presente edición en español, los tamaños y costos de la tabla 7.1 están totalmente desactualizados. Los 64 Mb de memoria cuestan menos de $ 20, siendo habitual el uso de 256 Mb en una máquina convencional. Análogamente, por $ 200.- se obtienen discos del orden de 40 GB o mejores.

Tipo de memoria	Tiempo de acceso	Costo por Mbyte	Tamaño típico utilizado	Costo aproximado
Registros	1 ns	Alto	1 Kb	-
Cache	5-20 ns	$ 100	1 Mb	$ 100
Memoria principal	60-80 ns	$ 1,10	64 Mb	$ 70
Discos	10 ms	$ 0,05	4 GB	$ 200

Tabla 7.1 • Propiedades de las distintas jerarquías de memoria (valores estimativos año 1999).

7.2 Memoria de acceso aleatorio

Esta sección analiza la estructura y funcionamiento de la **memoria de acceso aleatorio** (RAM, *random access memory*). En este contexto, el término **aleatorio** significa que puede accederse a cualquier celda de memoria en el mismo tiempo, independientemente de su posición en la estructura de la memoria.

La figura 7.2 muestra el comportamiento funcional de una celda de memoria RAM utilizada en una computadora típica. En la figura se representa al elemento de memoria como un *flip flop* D, con los controles necesarios para que la celda pueda ser seleccionada, leída y escrita. Existe una línea de datos (bidireccional) para la entrada y salida de los datos. Durante el análisis de los circuitos integrados de memoria RAM se utilizará un modelo de celda similar al que se muestra en la figura. Nótese que esta ilustración no representa necesariamente la implementación física real, sino solo su conducta funcional. Existen muchas formas de implementar una celda de memoria.

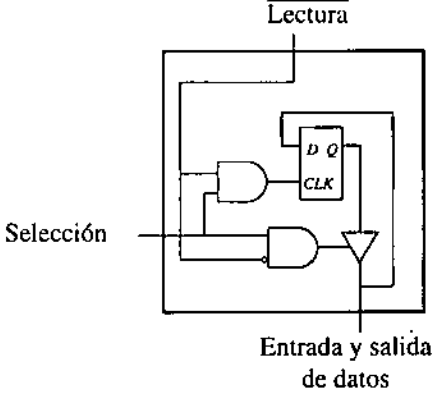


Figura 7.2 • Descripción funcional del comportamiento de una celda de memoria de acceso aleatorio.

Los circuitos de memoria de acceso aleatorio basados en *flip flops*, como el de la figura 7.2, se conocen como circuitos de **memoria estática** (SRAM, *static RAM*), debido a que el contenido de cada posición de la memoria se mantiene en tanto se mantenga la alimen-

tación eléctrica del circuito integrado. Los circuitos integrados de **memoria dinámica**, llamados **DRAM**, utilizan un **capacitor** que almacena una pequeña cantidad de carga eléctrica, y en el cual el nivel de carga representa un 0 o un 1. Los capacitores son mucho más chicos en tamaño que los *flip flops* y, por consiguiente, un circuito integrado de memoria dinámica basada en capacitores puede almacenar en la misma superficie una cantidad de información mucho mayor que una memoria estática. Dado que las cargas de los capacitores se van disipando en el tiempo, las memorias dinámicas requieren que la carga de sus celdas sea restablecida, o **refrescada**, en forma periódica y con frecuencia.

Las memorias dinámicas pueden sufrir descargas prematuras como resultado de la interacción con rayos gamma, cuya ocurrencia es natural. Este evento es estadísticamente raro, por lo que un sistema puede estar funcionando por días antes de que ocurra un error. Por esta razón, las primeras computadoras personales no utilizaban circuitos de detección de errores, dado que esas máquinas solían apagarse al final de cada día y, por lo tanto, había posibilidad de acumulación de errores no detectados. Esto permitió que los precios de las computadoras personales se mantuviesen en un rango competitivo. Como consecuencia de las drásticas disminuciones de precios de las memorias dinámicas, y de la creciente utilización de las computadoras personales en aplicaciones como cajeros automáticos y servidores de red, hoy es común la utilización de circuitos de detección de errores en las computadoras personales.

En la próxima sección se analiza la estructura de las celdas de memorias dinámicas de acceso aleatorio y cómo se las organiza dentro de un circuito integrado.

7.3 Organización de un circuito integrado

La figura 7.3 muestra un esquema simplificado de las conexiones de un circuito integrado de memoria RAM. En los terminales A_0-A_{m-1} se aplica una palabra de direcciones de m bits, formada por m líneas numeradas desde 0 hasta m-1. Simultáneamente, se activa la señal $\overline{CS}$ (*Chip Select*) junto con $\overline{WR}$ (para la escritura de un dato en memoria) o con WR (para la lectura de un dato desde la memoria). Las barras por encima de los símbolos CS y WR indican que el circuito integrado se selecciona cuando $CS = 0$ y que se producirá una operación de escritura cuando $WR = 0$. Cuando se lea información desde un circuito integrado, la palabra de datos de w bits aparecerá en las líneas de datos D_0-D_{w-1} luego de un período de tiempo T_{AA} (el retardo de tiempo medido desde el momento en que se validan las líneas de dirección hasta el instante en que se tienen los datos disponibles en la salida). Cuando se escriben datos en un circuito integrado, las líneas de datos también deben mantenerse estables (válidas) por un período de tiempo T_{AA}. Nótese que las líneas de datos de la figura 7.3 son bidireccionales, como ocurre normalmente en los sistemas de memoria.

Las líneas de dirección A_0-A_{m-1} del circuito integrado de la figura 7.3 forman una dirección, la que se decodifica a partir de una dirección de m bits a una de 2^m direcciones dentro del circuito integrado, cada una de las cuales se asocia con una palabra de w bits. El circuito integrado contiene, por lo tanto, 2^m x w bits.

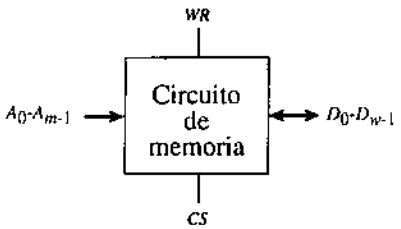


Figura 7.3 • Distribución simplificada de conexiones en un circuito integrado de memoria de acceso aleatorio.

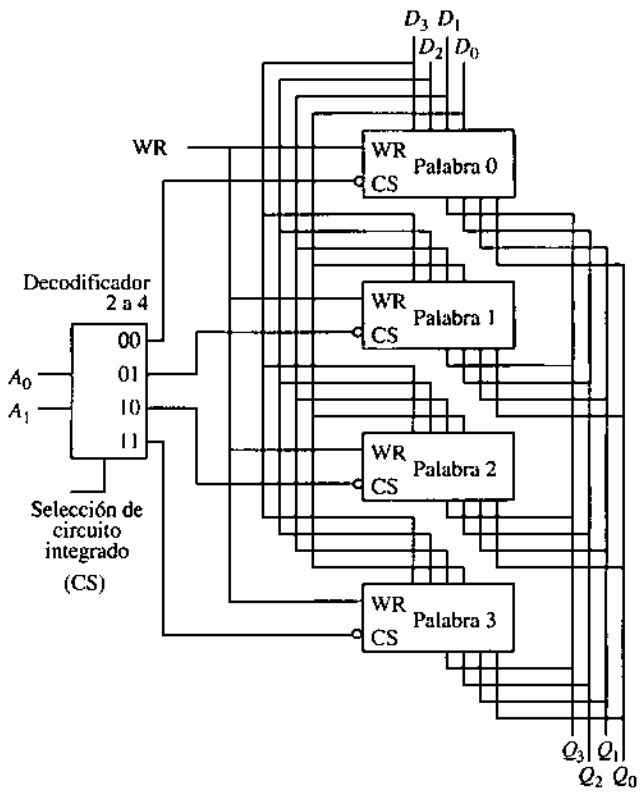


Figura 7.4 • Una memoria de cuatro palabras con cuatro bits por palabra y organización 2D.

Se analizará ahora el problema de crear una memoria RAM que almacene cuatro palabras de cuatro bits cada una. Una memoria de acceso aleatorio puede considerarse como una colección de registros. Se pueden usar cuatro registros para almacenar las palabras y, luego, se puede introducir un mecanismo de direccionamiento que permita la selección de una de las palabras para su lectura o escritura. En la figura 7.4 se muestra un diseño posible de esta memoria. Dos líneas A_0 y A_1 seleccionan la palabra a ser leída o escrita, a través del decodificador 2-4. Las salidas de los registros pueden interconectarse en forma segura sin el riesgo de un cortocircuito dado que el decodificador asegura que a lo sumo se

selecciona un registro por vez, y los registros deshabilitados se desconectan eléctricamente por medio del uso de *buffers* de tres estados. La línea de selección de componente (*Chip Select*) no hace falta en el decodificador, pero se la utiliza porque posteriormente será necesaria para la construcción de memorias más grandes. La figura 7.5 muestra un esquema simplificado de la memoria RAM.

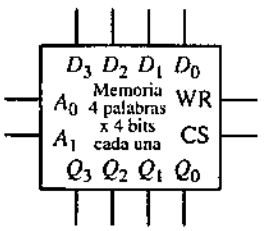


Figura 7.5 • Versión simplificada de la memoria RAM de cuatro palabras de cuatro bits.

Existen dos formas prácticas para organizar la memoria genérica de la figura 7.3. En los circuitos integrados más pequeños es habitual el uso de un solo decodificador para seleccionar una de 2^m palabras, cada una de las cuales contiene w bits. No obstante, esta organización no es económica en los actuales circuitos integrados de memoria. Debe tenerse en cuenta que un circuito integrado de 64 M x 1 requiere 26 líneas de direccionamiento (64 M = 2^{26}), lo que significa que un decodificador convencional requeriría 2^{26} compuertas Y de 26 entradas cada una, lo que representa un costo elevado en términos de superficie requerida, y esto tan solo para la decodificación.

Dado que la forma de la mayoría de los circuitos integrados es prácticamente cuadrada, una estructura de decodificación alternativa reduce significativamente la complejidad del decodificador al realizar las decodificaciones de las filas y de las columnas en forma separada. Esta organización se conoce como estructura 2-1/2D. La organización 2-1/2D es por lejos la preferida en el diseño de circuitos integrados de memorias RAM. La figura 7.6 ilustra la estructura de una memoria RAM de 2^6 palabras por un bit, con organización 2-1/2D. Las seis líneas de direcciones se dividen en forma pareja entre un decodificador de filas y un decodificador de columnas (el decodificador de columnas es, en realidad, la combinación de un multiplexor con un demultiplexor). Se utiliza una única línea bidireccional para la entrada y la salida de los datos.

Durante una operación de lectura, se selecciona una columna íntegra, la que se ingresa al multiplexor de columnas, el cual selecciona un único bit que será enviado al exterior de la memoria. Durante una operación de escritura, el demultiplexor de columnas orienta al único bit a ser escrito hacia la columna correspondiente, mientras que el decodificador de filas define la fila en la que deberá escribirse dicho bit.

En la práctica, para reducir la cantidad de terminales, el circuito integrado presenta solo $m/2$ terminales de direccionamiento; por consiguiente, las direcciones de fila y de columna se multiplexan en el tiempo sobre esos $m/2$ terminales de dirección. En este caso, se ingresa primero la dirección de fila, de $m/2$ bits, junto con una señal de sincronis-

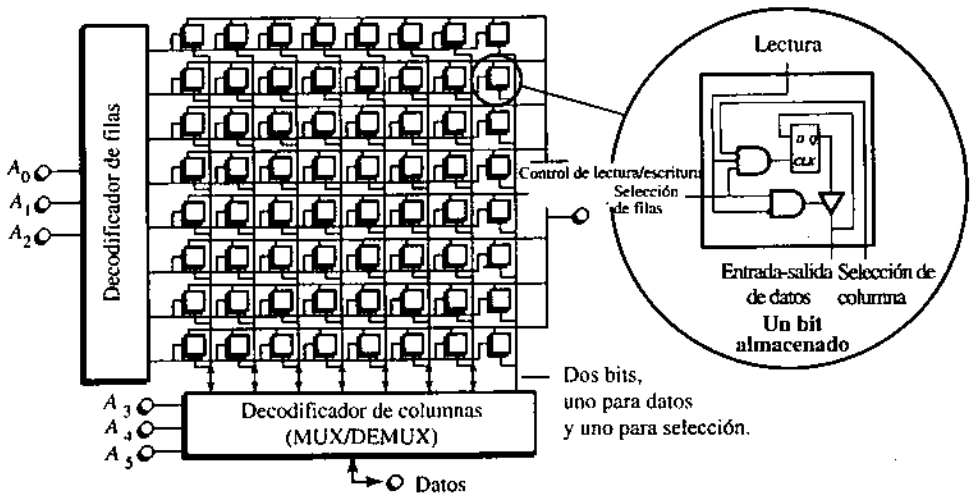


Figura 7.6 • Organización 2-1/2D de una memoria de 64 palabras por 1 bit.

mo de filas, $\overline{RAS}$ (*row address strobe*). La dirección de filas se almacena en el circuito integrado y se decodifica. Posteriormente, se ingresa la dirección de columnas, también de $m/2$ bits, junto con una señal de selección de columnas, $\overline{CAS}$ (*column address strobe*). Pueden existir otras terminales de entrada que sirvan para controlar las funciones de actualización del circuito integrado y otras funciones de la memoria.

Aun con esta organización 2-1/2D y fraccionando la dirección en componentes para las filas y las columnas, se necesita una cantidad importante de entradas y salidas sobre las compuertas lógicas del decodificador, y el número (aun grande) de terminales de dirección exige una gran cantidad de espacio para la conexión de los circuitos integrados de memoria sobre los circuitos impresos en los que van montados. Con el objeto de reducir todavía más los requerimientos de entradas y salidas, se pueden usar **árboles decodificadores**, los que se analizan en la sección 7.8.1. La sección 7.9 trata de una nueva arquitectura de memoria que ingresa las líneas de dirección en forma serie a través de una única terminal de entrada al circuito integrado.

Si bien las memorias RAM dinámicas son muy económicas, las memorias estáticas ofrecen mayor velocidad. Los ciclos de refresco, los circuitos de detección de errores y las bajas potencias de operación de las memorias dinámicas generan una diferencia de velocidades que es, aproximadamente, del orden de 1/4 de la velocidad de las memorias estáticas, pero las memorias estáticas también implican un costo significativo.

La eficiencia de ambos tipos de memorias (estáticas y dinámicas) puede mejorarse. Normalmente, se suele acceder en secuencia a un conjunto de palabras que constituyen un **bloque**. En esta situación, los accesos a memoria se pueden **entrelazar** de modo tal que mientras una memoria accede a la dirección A_m, otras memorias acceden a A_{m+1}, A_{m+2}, A_{m+3}, etc. De esta forma, el tiempo de acceso a cada palabra aparenta ser varias veces menor que el real.

7.3.1 Construcción de una memoria grande a partir de memorias pequeñas

Se pueden construir módulos de memoria RAM de mayor tamaño, partiendo de memorias de menor tamaño. Es posible el incremento tanto del tamaño de la palabra como de la cantidad de palabras por módulo. Por ejemplo, pueden combinarse ocho módulos de memoria de 16 M x 1 bit para formar un módulo de 16 M x 8 bits, así como pueden combinarse 32 módulos de 16 M x 1 bit para formar una memoria de 64 M x 8 bits.

Como ejemplo, considérese el circuito integrado de 4 palabras de 4 bits de la figura 7.5, el que será utilizado como bloque constructivo para formar, primero, un módulo de 4 palabras de 8 bits y, posteriormente, un módulo de 8 palabras de 4 bits. Se desea incrementar el tamaño de la palabra de cuatro bits y también el número de palabras. Considérese, primero, el problema de incrementar el ancho de la palabra desde cuatro bits a ocho. Esto se puede lograr en forma simple utilizando dos circuitos integrados, interconectando las líneas $\overline{CS}$ de ambos, para que sean seleccionados en forma conjunta, y conectando sus líneas de datos, en la forma en que se muestra en la figura 7.7.

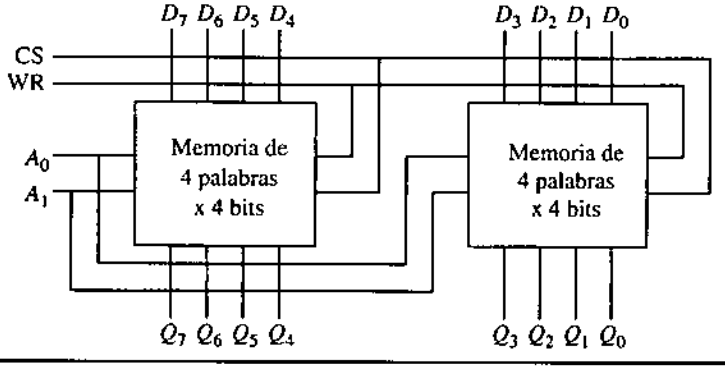


Figura 7.7 • Dos memorias de cuatro palabras por cuatro bits utilizadas para crear una memoria de cuatro palabras de ocho bits.

Si ahora se considera el problema de aumentar la cantidad de palabras de cuatro a ocho, la figura 7.8 muestra una configuración que resuelve este problema. Las ocho palabras se distribuyen entre las dos memorias de cuatro palabras. La línea de direcciones A_2 se hace necesaria dado que ahora hay que direccionar ocho palabras. Un decodificador para la línea A_2 selecciona, por medio del uso de las respectivas líneas de $\overline{CS}$, el módulo de memoria superior o el inferior, tras lo cual se decodifican las restantes líneas de direcciones (A_0 y A_1) dentro del módulo seleccionado. La correcta combinación de ambos planteos permite llevar tanto el tamaño de la palabra como el de la memoria hasta límites arbitrarios.

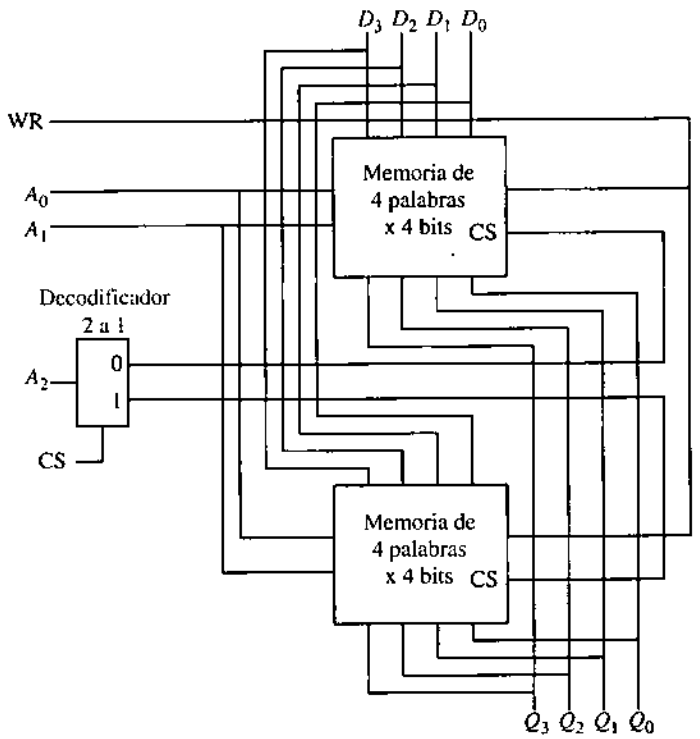


Figura 7.8 • Implementación de una memoria de ocho palabras de cuatro bits con dos memorias de cuatro palabras de cuatro bits cada una.

7.4 Módulos comerciales de memoria

Los circuitos integrados de memoria disponibles comercialmente suelen estar organizados en configuraciones normalizadas. La figura 7.9 muestra ocho circuitos integrados de 2^{20} bits, configurados en un módulo de memoria estructurado físicamente en una sola línea (SIMM, *single in line memory module*), que forman un módulo de memoria de 2^{20} x 8 bits (1 Mbyte). Los contactos eléctricos (numerados del 1 al 30) se hallan físicamente alineados. Para una memoria de 2^{20} posiciones se requieren 20 líneas de dirección, a pesar de lo cual el módulo solo provee 10 líneas de dirección (*A*0-*A*9). Las direcciones de fila y de columna, cada una de ellas de 10 bits, se cargan por separado, aplicándose las correspondientes señales de selección de dirección de columna (RAS) y de fila (CAS) una vez que el módulo recibe en forma válida la porción correspondiente de la dirección. Si bien esta organización parece duplicar el tiempo necesario para acceder a una dirección de memoria particular, en general el tiempo de acceso se mejora dado que solo puede hacer falta actualizar la dirección de fila o de columna.

Las ocho líneas de datos *DQ*1-*DQ*8 forman un byte que se lee o escribe en paralelo. Con el objeto de formar una palabra de 32 bits, se hacen necesarios cuatro módulos de tipo

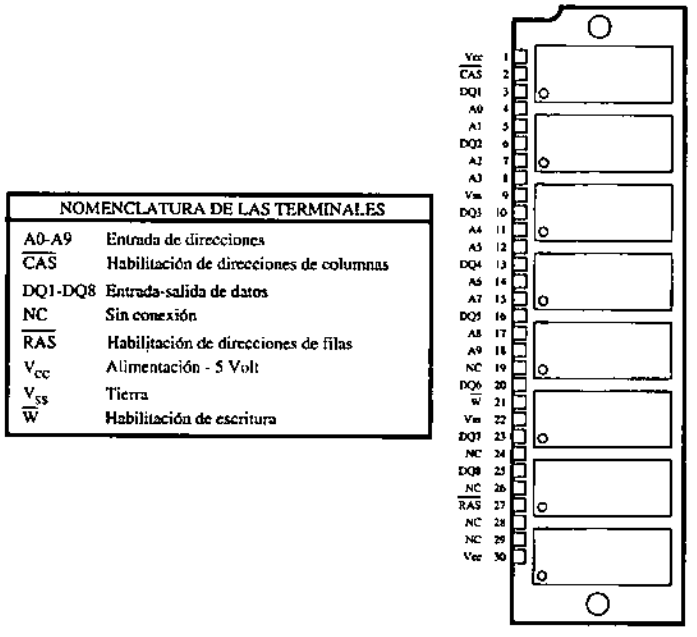


Figura 7.9 • Módulo de memoria SIMM (adaptado de Texas Instruments, 1991).

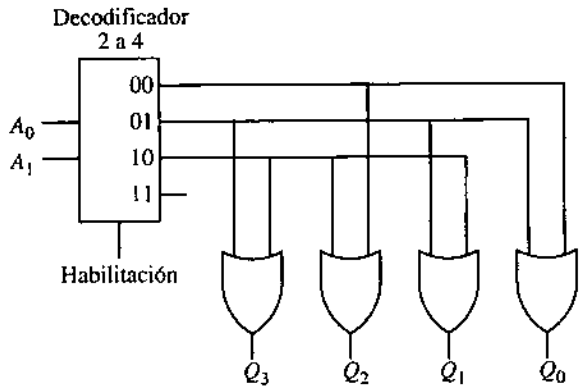
SIMM. Tal como en el caso de otras señales "activas en bajo", la línea de habilitación de escritura *Write Enable* se identifica con una barra encima de su símbolo ($\overline{W}$), lo que significa que la escritura se produce cuando se coloca un cero en la línea. En caso contrario se produce una lectura. La señal $\overline{RAS}$ también genera una operación de refresco, la que debe realizarse al menos cada 8 ms para permitir la recuperación de las cargas de los capacitores.

7.5 Memoria de lectura

Cuando un programa de computadora se carga en la memoria, se mantiene en la memoria hasta que se lo sobrescriba o hasta que se apague la energía eléctrica. Para algunas aplicaciones, el programa no cambia nunca, por eso se lo puede fijar en una **memoria de lectura** (ROM, *read only memory*). Las memorias de lectura se usan para almacenar programas en videojuegos, calculadoras, hornos de microondas y controladores de inyección de combustible, entre muchas otras aplicaciones.

La memoria de lectura es un dispositivo simple. Todo lo que se requiere es un decodificador, algunas líneas de salida y unas pocas compuertas lógicas. No hay necesidad de *flip flops* o capacitores. La figura 7.10 muestra una memoria de lectura en la que se almacenan cuatro palabras de cuatro bits (0101, 1011, 1110 y 0000). Cada entrada de direccionamiento (00, 01, 10, 11) corresponde a una de las palabras almacenadas.

En aplicaciones de gran volumen, las memorias de lectura se programan en el proceso de fabricación. Como alternativa, para aplicaciones de bajo volumen o para uso en



Posición	Palabra almacenada
00	0101
01	1011
10	1110
11	0000

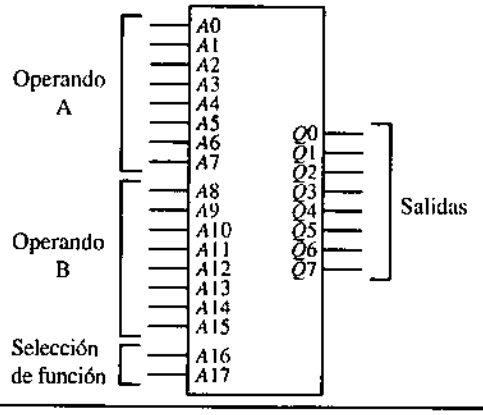
Figura 7.10 • Una memoria de lectura que almacena cuatro palabras de cuatro bits.

prototipos, se utilizan habitualmente memorias programables (PROM, *Programmable ROM*), las que permiten que el usuario genere sus contenidos por medio de un dispositivo de no muy alto precio conocido como **programador de PROM**. Desafortunadamente para las industrias que hacen uso de estas memorias, los programadores de memorias PROM también permiten su lectura, por lo que se puede duplicar el contenido de una memoria PROM en otro circuito integrado, o, lo que es peor aún, se puede descifrar su contenido haciendo un proceso de ingeniería inversa, para después modificar y reproducir el contenido de la memoria hacia nuevos circuitos de contrabando.

Si bien la memoria PROM permite que el diseñador demore la decisión acerca de los contenidos a almacenar en ella, este tipo de memoria solo puede grabarse una vez. En ciertos casos puede ser modificada, pero únicamente cuando el patrón de datos existente es un subconjunto del nuevo patrón de datos a grabar. Las memorias PROM borrables (EPROM) pueden rescribirse varias veces tras ser borradas con radiación ultravioleta (en el caso de las memorias UVPROM), a través de una ventana que se monta sobre la cápsula del circuito integrado. Las memorias PROM de borrado eléctrico (EEPROM) permiten que sus contenidos sean modificados eléctricamente.* Las nuevas memorias ***flash*** pueden rescribirse eléctricamente decenas de miles de veces, y se usan ampliamente en cámaras digitales de video y en programas de control de decodificadores televisivos, entre otras aplicaciones.

Las memorias PROM se utilizarán en secciones posteriores de este texto para integrar unidades de control y **unidades aritmético-lógicas**. Como ejemplo de este tipo de aplicación, considérese el diseño de una unidad aritmético-lógica que realice las cuatro ope-

* *N. de T.*: La diferencia entre las memorias UVPROM y EEPROM está dada por varias características. No solo es diferente el proceso de borrado (en un caso mediante radiación ultravioleta, en el otro vía corriente eléctrica), sino que también son distintos los procesos de escritura, dado que las UVPROM, al igual que las PROM se graban en un dispositivo específico, en tanto que las EEPROM pueden grabarse sin ser extraídas del circuito en que se encuentran. Por otra parte, la vida útil, en ciclos de escritura, de las EEPROM es muy inferior a la de las UVPROM, por lo que sus ámbitos de aplicación suelen ser totalmente diferentes.



A17	A16	Función
0	0	Sumar
0	1	Restar
1	0	Multiplicar
1	1	Dividir

Figura 7.11 • Una tabla de búsquedas que implementa una unidad aritmético-lógica de ocho bits.

raciones aritméticas básicas, suma, resta, producto y cociente, sobre datos de ocho bits, operando en módulo 256 (salida de 8 bits). Se puede generar una tabla de verdad que enumere todas las 2^{16} posibles combinaciones de los operandos y las 2^2 combinaciones de funciones, y colocar esa tabla de verdad en una memoria PROM por medio de su grabación en un dispositivo programador.

Esta solución de fuerza bruta, que implementa una **tabla de búsquedas**, no es tan poco práctica como puede parecer y se utiliza en una gran cantidad de situaciones. La memoria no requiere tener un tamaño muy grande. Hay $2^8 \times 2^8$ combinaciones de los dos operandos de entrada, y hay 2^2 funciones, por lo tanto la memoria PROM requiere un total de $2^8 \times 2^8 \times 2^2 = 2^{18}$ palabras, lo que a los efectos prácticos representa una memoria pequeña. La configuración de la unidad aritmético-lógica implementada por medio de la memoria PROM se muestra en la figura 7.11. Las líneas de direcciones se utilizan para los operandos y para las entradas de selección de función, en tanto que las salidas se generan simplemente recuperando el valor previamente calculado y almacenado en la posición direccionada. Este enfoque es más rápido que la implementación circuital de las funciones, pero no puede extenderse a palabras formadas por muchos bits sin aplicar algún tipo de descomposición. Las computadoras de hoy trabajan en forma habitual con operandos de 32 y más bits, con lo que una unidad aritmético-lógica implementada con una PROM requeriría $2^{32} \times 2^{32} \times 2^2 = 2^{66}$ palabras, lo que aún es prohibitivamente grande.

7.6 Memoria cache

Cuando se ejecuta un programa en una computadora, la mayor parte de las referencias de memoria se hacen con respecto a una pequeña cantidad de direcciones. En general, el 90% del tiempo de ejecución se consume en aproximadamente el 10% del código. Esta propiedad se conoce como **principio de localidad**. Cuando un programa hace referencia a una locación de memoria, muy probablemente vaya a acceder de nuevo a ella en el corto pla-

zo, lo que se conoce como **localidad temporal**. En forma similar, existe una **localidad espacial**, en la cual se plantea que tras una referencia a una posición de memoria dada, es mucho más probable que se acceda a posiciones cercanas a ella que a posiciones lejanas a la misma. La localidad temporal se produce debido a que los programas, en general, consumen mucho tiempo en iteraciones o en actividades recursivas, por lo que recorren la misma sección de código una enorme cantidad de veces. La localidad espacial se deriva de la tendencia a almacenar los datos en zonas contiguas. Si bien el planteo referido al 10% del código surge del total de las referencias a memoria, dentro de ese 10% los accesos tienden a estar agrupados. Por consiguiente, para un intervalo de tiempo dado, la mayor parte de los accesos a memoria se producen en un conjunto de posiciones aún menor al 10% del tamaño del programa.

El acceso a memoria suele ser lento en comparación con la velocidad de la unidad central de proceso, por lo que la memoria genera un cuello de botella importante en el rendimiento de la computadora. Dado que la mayoría de las referencias de memoria provienen de un conjunto pequeño de locaciones, se puede aprovechar el principio de localidad para mejorar dicho rendimiento. Con este objetivo se puede colocar entre la memoria principal y la unidad central de procesos una **memoria *cache***, pequeña pero rápida, con el objeto de almacenar los contenidos de las direcciones a las que se accede con mayor frecuencia. Durante la ejecución de un programa, se analiza primero el contenido de la memoria *cache*, y se accede a la palabra requerida si estuviese presente en la misma. Si la palabra a la que se hace referencia no está presente en la memoria *cache*, se genera una posición vacía y se carga la palabra requerida en esa posición desde la memoria principal, tras lo cual se accede en la memoria *cache* a la palabra solicitada. Si bien este proceso lleva más tiempo que el acceso directo a la memoria principal, el rendimiento general mejora cuando se logra que una proporción alta de accesos a la memoria se satisfaga desde la memoria *cache*.

Los sistemas modernos de memoria pueden tener distintos niveles de memoria *cache*, a los que se suele distinguir como nivel 1 (L1), nivel 2 (L2) y, aún en ciertos casos, nivel 3 (L3). En muchas implementaciones, la memoria *cache* de nivel 1 viene incorporada directamente en el circuito integrado de la CPU. Entre los procesadores comerciales, tanto el Pentium Intel como el PowerPc G3 de Motorola-IBM incluyen 32 Kbytes de **memoria *cache*** de nivel 1 en el circuito integrado del procesador.

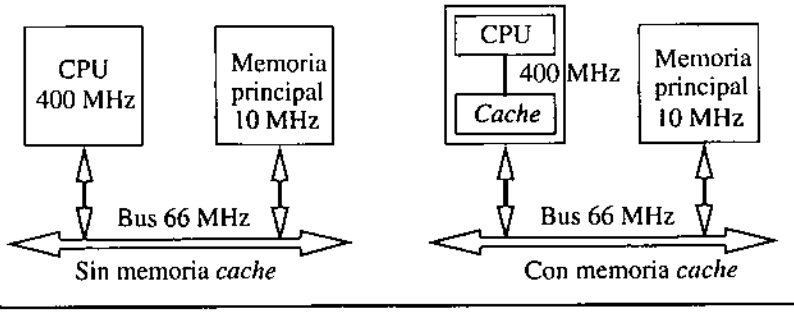


Figura 7.12 • Ubicación de una memoria *cache* en un sistema de computación.

La memoria *cache* es más veloz que la memoria principal debido a una cantidad de razones diferentes. Se puede utilizar una electrónica más rápida, lo que implica también un costo mayor en términos monetarios, de tamaño y de requerimientos de alimentación eléctrica. Dado que la memoria *cache* es pequeña, este incremento de costos es, a su vez, relativamente moderado. Una memoria *cache* tiene menor cantidad de direcciones que la memoria principal y, como resultado, tiene un árbol de decodificación poco profundo, lo que reduce el tiempo de acceso. Además, la memoria *cache* se encuentra ubicada, tanto física como lógicamente, más cerca de la CPU que la memoria principal, lo que evita los retardos en las transferencias sobre un **bus compartido**.

La figura 7.12 ilustra una situación típica. A la izquierda de la figura se muestra una computadora simple, sin memoria *cache*. Esta computadora posee una CPU con una frecuencia de reloj de 400 MHz, pero se comunica a través de un bus de 66 MHz, con una memoria principal que soporta una frecuencia máxima de reloj de 10 MHz. La sincronización de la CPU con el bus requiere normalmente algunos pocos ciclos de reloj; por consiguiente, la diferencia de velocidades entre la memoria principal y la CPU puede estar en el orden de 10 veces o mayor. Tal como se muestra a la derecha en la figura 7.12, se puede intercalar una memoria *cache* cercana a la CPU, de modo tal que la unidad de proceso pueda manejar accesos rápidos sobre un trayecto directo a la memoria *cache* a 400 MHz.

7.6.1 Memoria cache de asignación asociativa

El proceso de asignación de las direcciones de memoria principal a las direcciones de memoria *cache* ha llevado al desarrollo de distintos mecanismos de hardware. El usuario no necesita conocer el mecanismo de conversión, lo que tiene la ventaja de permitir la introducción de las mejoras en la memoria *cache* sin la necesidad adicional de modificar los programas de aplicación.

Los diferentes métodos de asignación de memoria *cache* influyen sobre el costo y el rendimiento, y no hay un único método que resulte ser el mejor y el más adecuado para todas las situaciones. En esta sección se analiza el esquema de asignación **asociativa**. En la figura 7.13 se ilustra un esquema asociativo para un espacio de memoria de 2^{32} palabras, divididas en 2^{27} **bloques** de 32 palabras por bloque. La memoria principal no está físicamente dividida de esta manera, pero esta es la visión que tiene la memoria *cache* de la memoria principal. Los bloques de memoria *cache*, o **líneas de *cache***, como también se las conoce, tienen tamaños que van desde 8 hasta 64 bytes. La información ingresa y egresa de la memoria ***cache*** de a una línea por vez, utilizando las técnicas de entrelazado de memoria analizadas con anterioridad.

La memoria ***cache*** de este ejemplo consiste en 2^{14} **líneas** en las que se pueden colocar los bloques de memoria principal. Existen más bloques de memoria principal que líneas de memoria *cache*, y cualquiera de los 2^{27} bloques de memoria principal puede colocarse en cualquier línea de la memoria *cache* (a razón de un único bloque en cada línea en un mo-

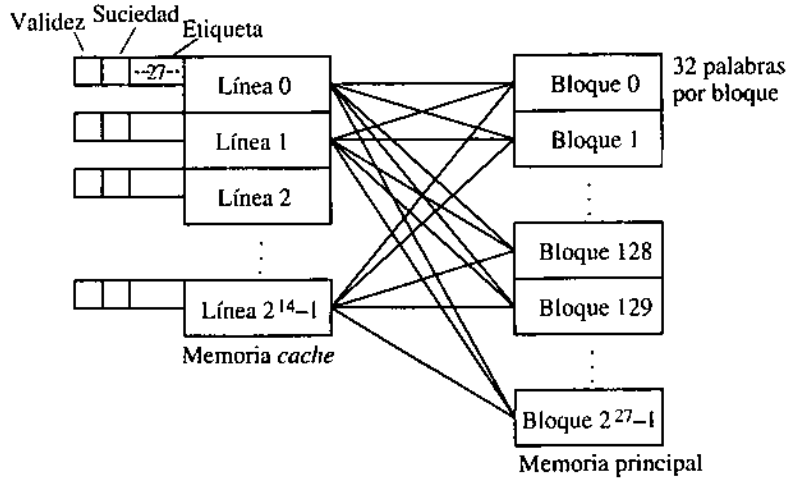


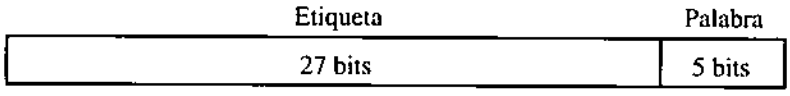
Figura 7.13 • Un esquema de asignación asociativa para una memoria *cache*.

mento dado de tiempo). Para mantener la pista acerca de cuál de los posibles 2^{27} bloques de memoria se encuentra en cada línea, se agrega a cada una de ellas un campo de identificación o **etiqueta** (*tag*), de 27 bits, la que lleva un identificador cuyo rango va desde 0 hasta $2^{27} - 1$. El campo de etiqueta se ubica en los 27 bits más significativos de los 32 bits presentados a la memoria *cache* como dirección. Todas las etiquetas se almacenan en una memoria especial, en la que se los puede buscar en paralelo. Cada vez que se almacena un bloque nuevo en la memoria *cache*, su etiqueta se almacena en la posición correspondiente de la memoria de etiquetas.

Cuando se carga por primera vez un programa en la memoria principal, se limpia la memoria *cache* y, en consecuencia, durante la ejecución de un programa se requiere un **bit de validez** para indicar si la línea contiene un bloque que pertenece o no al programa que se está ejecutando. Asimismo, existe un **bit de suciedad** (*dirty bit*) que controla si se ha producido la modificación de algún bloque durante su almacenamiento en la memoria *cache*. Si una línea se modifica, deberá ser reescrita en la memoria principal antes de que sea utilizada nuevamente con otro bloque.

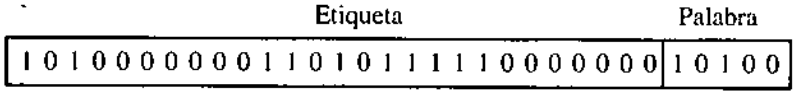
Una referencia a una locación que se encuentra en la memoria *cache* da por resultado un **acierto**; en caso contrario, el resultado es una **falla**. Cuando se carga por primera vez un programa en la memoria, los bits de validez se colocan todos en cero. Por consiguiente, la primera instrucción a ser ejecutada provocará una falla de la memoria *cache*, dado que hasta ese momento no habrá absolutamente nada del programa en la memoria *cache*. Se ubica en memoria principal el bloque que causó la falla y se lo coloca en la memoria *cache*.

En una memoria *cache* de asignación asociativa, cada bloque de memoria principal puede asignarse a cualquier línea. La asignación de los bloques de memoria principal se realiza por medio de la partición de la dirección en campos, uno para la etiqueta y otro para la palabra (también conocido como campo "byte") en la forma siguiente:



Cuando se hace referencia a una dirección de memoria principal, la electrónica de la memoria *cache* intercepta esa referencia y busca dentro de la memoria de etiquetas para ver si el bloque requerido se encuentra almacenado en la memoria *cache*. Para cada una de las líneas, se verifica si su bit de validez es 1, y en caso afirmativo, se compara el campo de etiqueta de la dirección buscada con el campo de etiqueta de la línea. Todas las etiquetas se analizan en paralelo, usando una **memoria asociativa** (que es algo diferente de un esquema de asignación asociativa; en la sección 7.8.3 se analizan las memorias asociativas). Si alguna etiqueta de la memoria de etiquetas coincide con el campo de etiquetas de la referencia a memoria principal, se extrae la palabra de la línea desde la posición indicada por el campo de palabra. Si la palabra a que se hizo referencia no se encuentra en la memoria *cache*, se deberá traer desde la memoria principal el bloque de memoria que contiene la palabra buscada, recuperando luego esa palabra referida desde la memoria *cache*. Se actualizan los campos de etiqueta, validez y suciedad, y se retoma la ejecución del programa.

A continuación se describe la forma en que se procede a asignar a la memoria *cache* el acceso a la dirección de memoria $(A035F014)_{16}$. Los 27 bits más significativos de la dirección forman el campo de etiquetas, en tanto que los cinco bits restantes forman el campo de palabra, según se indica:



Si la palabra direccionada está en la memoria *cache*, se la encontrará en la posición $(14)_{16}$ de una línea cuya etiqueta es $(501AF80)_{16}$, obtenida con los 27 bits más significativos de la dirección. Si la palabra direccionada no se encuentra en la memoria *cache*, se copiará el bloque correspondiente al campo de etiquetas $(501AF80)_{16}$ desde la memoria principal a alguna de las líneas disponibles en la memoria *cache*, con lo que luego podrá satisfacerse desde la memoria *cache* la referencia a memoria que provocara la falla.

Si bien este esquema de asignación es lo suficientemente poderoso como para satisfacer una amplia gama de situaciones de acceso a memoria, existen dos problemas de implementación que limitan su rendimiento. En primer lugar, el proceso de decodificar cuál de las líneas debe liberarse cuando se trae un nuevo bloque hacia la memoria *cache* puede ser complejo. Este proceso requiere una cantidad importante de electrónica, además de introducir retardos en los accesos a memoria. Un segundo problema surge del hecho de que la búsqueda en la memoria *cache* requiere la comparación del campo de etiquetas con la totalidad de las etiquetas (2^{14}) de la misma. En las secciones 7.6.2 y 7.6.3 se describen métodos alternativos que permiten reducir el número de comparaciones requeridas.

Políticas de reemplazo en memorias de asignación asociativa

Cuando se requiere almacenar un bloque nuevo de memoria principal en una memoria *cache* de asignación asociativa, se debe identificar alguna línea disponible para el almacenamiento. Si hay líneas sin uso, tal como ocurre cuando se inicia la ejecución de un programa, se puede usar simplemente la primera línea cuyo campo de validez sea 0. Cuando todos los bits de validez de todas las líneas de la memoria *cache* valen 1, debe liberarse alguna de las líneas activas para permitir cargar el nuevo bloque. Existen cuatro políticas o criterios de reemplazo que se utilizan con mayor frecuencia, las que se conocen respectivamente como criterio del elemento **menos recientemente usado** (LRU, *least recently used*), criterio de "**el primero que entra es el primero que sale**" (FIFO, *first in first out*), criterio del **menos frecuentemente usado** (LFU, *least frequently used*) y un último criterio **aleatorio**. Una quinta técnica que se utiliza solo con propósitos de análisis es la del reemplazo **óptimo**.

Para el criterio LRU de reemplazo del elemento usado menos recientemente se agrega en cada línea una identificación de tiempo, la que se actualiza cada vez que se accede a la línea. Cuando se debe liberar una línea para incorporar un nuevo bloque, se descarta el contenido de la línea usada menos recientemente, la que se reconoce a través de la edad de su identificación temporal, utilizándose esa línea para almacenar el nuevo bloque. El esquema LFU, que reemplaza la línea menos frecuentemente usada, funciona en forma similar, excepto por el hecho de que se actualiza una línea por vez mediante un contador de frecuencia adosado a cada línea. Cuando se requiere cargar un bloque nuevo, se libera aquel que muestra la menor tasa de utilización. La política FIFO reemplaza líneas en forma ordenada, una tras otra en el orden en que se las encuentra físicamente ubicadas dentro de la memoria *cache*. La técnica aleatoria simplemente elige una línea al azar.

La política del reemplazo óptimo no es práctica pero se usa con propósitos comparativos para determinar la eficiencia de las otras técnicas de reemplazo con respecto a la mejor posible. Esto significa que solo se puede determinar la política optima de reemplazo una vez que el programa ha sido efectivamente ejecutado, por lo que su utilidad en la ejecución de un programa es mínima.

A partir de estudios realizados, se puede determinar que la técnica del reemplazo del elemento menos frecuentemente usado (LFU) es solo ligeramente mejor que la política aleatoria. La técnica LRU se suele preferir debido a la posibilidad de su implementación eficiente. La sección 7.6.7 analiza una implementación simple de la técnica de reemplazo de la línea menos recientemente utilizada.

Ventajas y desventajas de la asignación asociativa de la memoria cache

La técnica de asignación asociativa tiene la ventaja de que cualquier bloque de memoria puede colocarse dentro de cualquier línea de la memoria *cache*. Esto significa que, sin interesar qué tan irregulares sean las referencias a programa y a datos, si se dispone de una

línea para el bloque, el mismo puede almacenarse en la memoria *cache*. Por consiguiente, se hace necesaria una cantidad considerable de electrónica para la administración y el control de la memoria *cache*. Cada línea debe tener una etiqueta de 27 bits que identifique su posición dentro de la memoria principal, y las etiquetas deben buscarse en paralelo. Considerando el ejemplo precedente, la memoria de etiquetas debe tener un tamaño de 27 x 2^{14} bits; además, como se ha dicho, se requiere el mecanismo que permita la búsqueda en paralelo de la memoria de etiquetas. Las memorias en las que puede realizarse la búsqueda de contenidos en paralelo se conocen como **memorias asociativas** o **direccionables por contenido**. El análisis de este tipo de memorias se reserva para una sección posterior de este capítulo.

Si se restringe la posición de memoria *cache* en la que se puede ubicar cada uno de los bloques de memoria principal, es posible eliminar la necesidad de usar una memoria asociativa. Este tipo de memoria *cache* se conoce como de **asignación directa** y se analiza en la sección siguiente.

7.6.2 Memoria cache de asignación directa

La figura 7.14 muestra un esquema de asignación directa para una memoria de 2^{32} palabras. Al igual que antes, la memoria se divide en 2^{27} bloques de $2^5 = 32$ palabras por bloque, y la memoria *cache* consiste en 2^{14} líneas. Hay muchos más bloques de memoria principal que líneas de memoria *cache*, por lo que se pueden asignar $2^{27}/2^{14} = 2^{13}$ bloques de memoria principal a cada línea de la memoria *cache*. Con el propósito de mantener el control de cuál de los 2^{13} bloques posibles se encuentra en cada línea, se agrega un campo de etiquetas de 13 bits, el que contiene un identificador cuyo rango va desde 0 hasta $2^{13} - 1$.

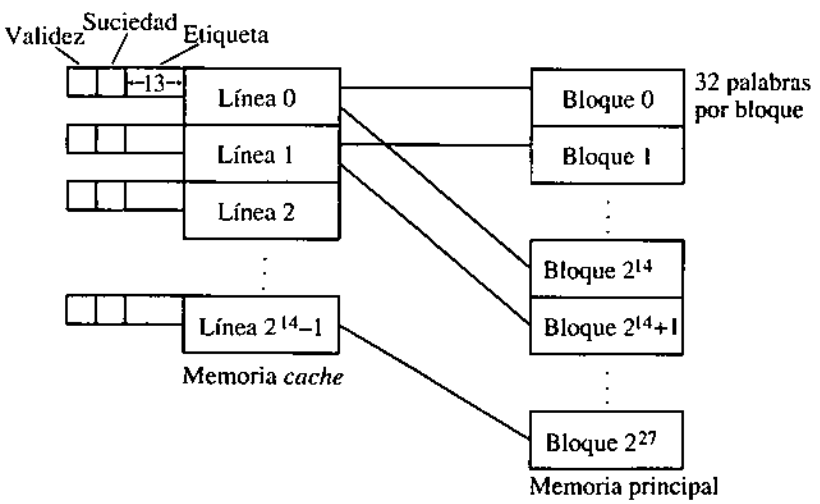
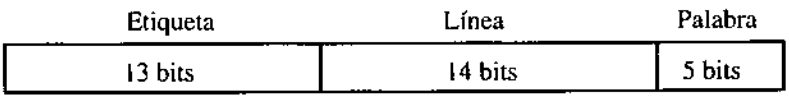


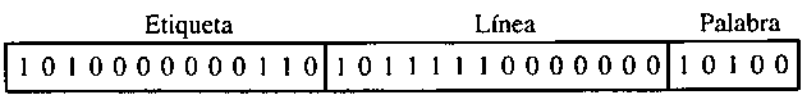
Figura 7.14 • Un esquema de asignación directa para una memoria *cache*.

El esquema se conoce como de "asignación directa" debido a que cada línea de memoria *cache* se corresponde con un conjunto explícito de bloques de memoria principal. En una memoria *cache* de asignación directa, cada bloque de memoria principal se asigna a una única línea, pero cada línea puede recibir más de un bloque. La asignación de los bloques de memoria principal a las líneas de la memoria *cache* se realiza a través de la partición de la dirección en campos asignados a la etiqueta, la línea y la palabra, en la forma siguiente:



La dirección de memoria principal, de 32 bits, se parte en un campo de etiqueta de 13 bits, seguido de otro campo de 14 bits que define la línea y este, a su vez, seguido por un campo de palabra de 5 bits. Cuando se hace referencia a una dirección de memoria principal, el campo de línea identifica en cuál de las 2^{14} líneas puede encontrarse el bloque, si es que estuviese dentro de la memoria *cache*. Si el bit de validez es 1, se compara el campo de etiquetas de la dirección de referencia con el campo de etiquetas de la línea. Si los campos de etiqueta coinciden, la palabra se toma desde la posición de la línea especificada en el campo de palabra. Si el bit de validez es 1 pero los campos de etiqueta no coinciden, se vuelve a escribir la línea a memoria principal si el bit de suciedad vale 1, y se lee el nuevo bloque desde la memoria principal hacia la línea de la memoria *cache*. Para un programa que recién se inicia, el bit de validez valdrá 0, por lo que el bloque directamente se escribe en la línea. Se coloca el bit de validez de esa línea en 1 y se reinicia la ejecución del programa.

Se considerará ahora, como ejemplo, la forma en que se asigna en la memoria *cache* el acceso a la dirección $(A035F014)_{16}$ de memoria principal. La palabra binaria se divide según el formato de palabra ilustrado anteriormente. Los 13 bits más significativos forman el campo de etiqueta, los 14 bits siguientes forman el campo de línea y los últimos cinco bits forman el campo de palabra, según se muestra a continuación:



Si la palabra buscada está en la memoria *cache*, se encontrará en la posición $(14)_{16}$ de la línea $(2F80)_{16}$, la cual tendrá a su vez una etiqueta de $(1406)_{16}$.

Ventajas y desventajas de la asignación directa de memoria cache

El esquema de asignación directa de memoria *cache* es relativamente simple de implementar. La memoria de etiquetas en el ejemplo anterior es de solo 13×2^{14} bits de tamaño, o sea, menos de la mitad del tamaño de la memoria de asignación asociativa. Más aún, no hay necesidad de una búsqueda asociativa, dado que el campo de línea de la dirección de memo-

ria principal presentada por la CPU se utiliza para dirigir la comparación hacia la única línea en que se podría encontrar el bloque, si realmente estuviese en la memoria *cache*.

Esta simplicidad tiene un costo. Considérese qué ocurre cuando un programa hace referencia a posiciones de memoria separadas por 2^{19} palabras, lo que coincide con el tamaño de la memoria *cache*. Este patrón puede aparecer en forma natural si se almacena una matriz en memoria por filas y se la accede por columnas. Cada referencia a memoria dará por resultado una falla, lo que obligará a leer todo un bloque hacia la memoria *cache* aún cuando solo haga falta usar una única palabra. Peor aún, se utilizará en realidad nada más que una pequeña fracción de la memoria *cache* disponible.

Se puede ver que cualquier programador que escriba un programa de esta manera merece obtener el peor rendimiento posible, pero, de hecho, las operaciones rápidas sobre matrices utilizan dimensiones que son potencia de dos (esto posibilita el empleo de instrucciones de desplazamiento para reemplazar las instrucciones más costosas de multiplicación y división que se requieren para la indexación de matrices). De tal manera, el peor escenario referido al acceso de posiciones de memoria que difieren en 2^{19} direcciones no es tan improbable. Para evitar esta situación sin pagar el alto precio de implementación de una **matriz totalmente asociativa**, se puede utilizar un esquema de asignación **asociativa por conjuntos**, el que combina aspectos de las asignaciones directa y asociativa. El método de asignación directa por conjuntos se describe en la sección siguiente.

7.6.3 Asignación asociativa por conjuntos de la memoria cache

El esquema de asignación asociativa por conjuntos combina la simplicidad de la asignación directa con la flexibilidad de la asignación asociativa. La asignación asociativa por conjuntos es más práctica que la asignación completamente asociativa debido a que la porción asociativa queda limitada a solo algunas líneas que forman un conjunto, tal como se muestra en la figura 7.15. En este ejemplo, dos bloques forman un conjunto, por lo que se tiene una memoria *cache* asociativa de **dos vías**. Si hay cuatro bloques en cada conjunto, se tendrá una memoria *cache* asociativa de cuatro vías.

Dado que hay 2^{14} líneas en la memoria *cache*, existen $2^{14}/2 = 2^{13}$ conjuntos. Para asignar una dirección a un conjunto se utiliza un esquema de asignación directa, en tanto que, dentro del conjunto, se utiliza una asignación asociativa. El formato de la dirección tiene 13 bits en el campo de conjunto, lo que identifica al conjunto en el que se encontraría la palabra a la que se quiere acceder si estuviese en la memoria *cache*. El campo de palabra tiene cinco bits, al igual que antes, y queda un campo de etiquetas de 14 bits, el que completa los 32 bits de la dirección, tal como se muestra a continuación:

Etiqueta	Conjunto	Palabra
14 bits	13 bits	5 bits

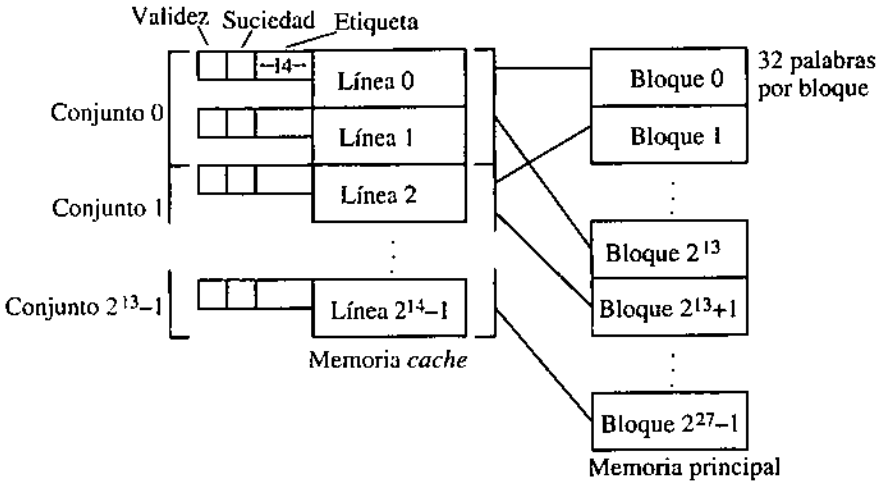


Figura 7.15 • Un esquema de asignación asociativa por conjuntos para una memoria *cache*.

Para ejemplificar cómo se ve una dirección de memoria principal desde una memoria *cache* asociativa por conjuntos, considérese nuevamente la dirección $(A035F014)_{16}$. Los 14 bits más significativos forman el campo de etiqueta, el que viene seguido por 13 bits del campo de conjunto y por otros 5 bits para el campo de palabra, de acuerdo con la siguiente estructura:

Etiqueta	Conjunto	Palabra
1 0 1 0 0 0 0 0 0 0 1 1 0 1	0 1 1 1 1 1 0 0 0 0 0 0 0	1 0 1 0 0

Al igual que antes, solo la memoria *cache* conoce la forma en que se divide el campo de direcciones, con lo que las traducciones de direcciones resultan abstractas para el resto de la computadora.

Ventajas y desventajas de la asignación asociativa por conjuntos

En el ejemplo precedente, la memoria de etiquetas se incrementa en tamaño solo ligeramente con respecto al ejemplo de la asignación directa, a 13 x 2^{14} palabras, y en cada referencia de memoria solo deben analizarse dos etiquetas. El esquema de memoria *cache* asociativa por conjuntos se utiliza en los procesadores actuales en forma casi universal.

7.6.4 Rendimiento de la memoria cache

Nótese que se puede reemplazar rápidamente la electrónica requerida para la asignación directa de memoria *cache* por electrónica relacionada con la asignación asociativa, sin necesidad de hacer otras modificaciones en la computadora o en sus programas. Al cambiar de método solo cambia el rendimiento del tiempo de ejecución.

El rendimiento durante el tiempo de ejecución es el objetivo planteado por el uso de una memoria *cache*, existiendo una cantidad de cuestiones que deben analizarse con respecto a las causas que conducen a provocar la transferencia de una palabra o bloque entre la memoria *cache* y la memoria principal. Las políticas de lectura y escritura de la memoria *cache* se resumen en la figura 7.16. Estas políticas dependen de si la palabra buscada se halla o no en la memoria *cache*. Si se está llevando a cabo una operación de lectura en la memoria *cache*, y la palabra requerida se encuentra en dicha memoria, se está en presencia de un acierto de la memoria *cache*, con lo que la palabra buscada se transfiere inmediatamente a la CPU. Cuando ocurre una falla de memoria *cache*, se produce la lectura hacia la memoria *cache* del bloque completo de memoria principal que contiene la palabra requerida.

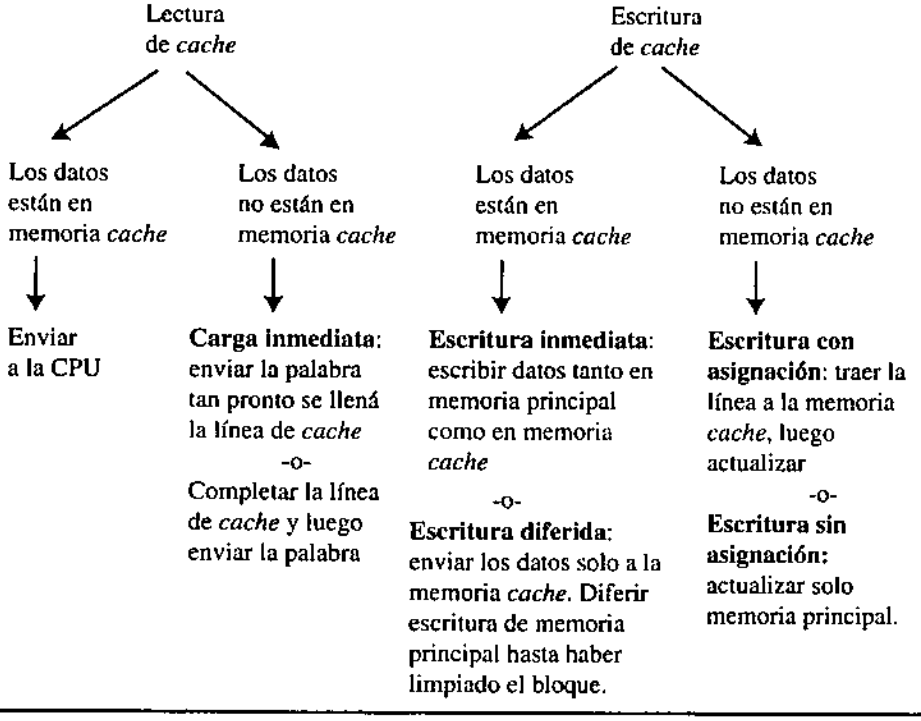


Figura 7.16 • Políticas de lectura y escritura de memoria *cache*.

En algunas organizaciones de memoria *cache*, la palabra que produce una falla se transfiere a la CPU ni bien se la lee hacia la memoria *cache*, en lugar de esperar a que se complete la transferencia del resto de la línea. Este esquema se conoce como **operación de carga inmediata** (*load-through*). Si la memoria principal no utiliza técnicas de entrelazado, y si la palabra requerida corresponde a la última posición del bloque, no se consigue ninguna mejora de eficiencia porque antes de transferir la palabra a la CPU ya se ha producido la transferencia de toda la línea de memoria *cache*. Si la memoria principal es entrelazada, puede organizarse el criterio de direccionamiento para que las transferencias hacia la CPU siempre produzcan una mejora en el rendimiento.

En las operaciones de escritura, si la palabra está en la memoria *cache*, puede haber dos copias de la palabra, una en la memoria *cache* y otra en la memoria principal. Si ambas se actualizan en forma simultánea, se produce lo que se conoce como una operación de **escritura inmediata** (*write through*). Si se difiere la escritura hasta después de la limpieza total de la línea de memoria *cache*, la operación se define como una operación de **escritura diferida** (*write back*). Aunque el elemento de datos no se encuentre en la memoria *cache* cuando se produce el proceso de escritura, existe la posibilidad de ingresar en la memoria *cache* el bloque que contiene la palabra y actualizarlo, lo que se conoce como un proceso de **asignación y escritura** (*write-allocate*), o de actualizarlo en memoria principal sin involucrar en el proceso a la memoria *cache*, lo que se identifica como una **escritura sin asignación** (*write-no-allocate*).

Algunas computadoras ofrecen memoria *cache* separada para instrucciones y datos, conocida como **memoria *cache* fraccionada**, lo que representa una variante de la **arquitectura Harvard**, en la que las instrucciones y los datos se almacenan en secciones diferentes de la memoria. Dado que las líneas correspondientes a las instrucciones nunca pueden estar sucias (a menos que el código sea del tipo automodificable, lo que no suele ser frecuente), la memoria *cache* para las instrucciones es más simple que la memoria *cache* para los datos. En apoyo de esta configuración, los estudios realizados muestran que la mayor parte del tráfico de memoria sale de la memoria principal. Estadísticamente, hay solo una operación de escritura de memoria por cada cuatro operaciones de lectura. Esta afirmación se justifica si se piensa que las instrucciones que se ejecutan en un programa solo se leen desde la memoria principal y no se escriben nunca excepto cuando lo hace el cargador del sistema. Además, las operaciones normalmente involucran la lectura de dos operandos y el almacenamiento de un único resultado, lo que significa que hay dos operaciones de lectura por cada operación de escritura. Una memoria *cache* que solo maneje lecturas y que transfiera las escrituras directamente a la memoria principal también puede ser efectiva, aunque no tan efectiva como una memoria *cache* totalmente funcional.

No es sencillo definir cuáles son las mejores técnicas de lectura y escritura de una memoria *cache*. La organización de la memoria *cache* se optimiza para cada arquitectura de computadora y para la combinación de programas que la computadora ejecuta. La organización y los tamaños de la memoria *cache* se determinan normalmente a partir de los resultados de procesos de simulación que verifican la naturaleza del tráfico de memoria.

7.6.5 Tasas de acierto y tiempos de acceso

Dos valores que caracterizan el rendimiento de una memoria *cache* son la **tasa de aciertos** y el **tiempo efectivo de acceso**. La tasa de aciertos se calcula dividiendo la cantidad de veces en que se han encontrado efectivamente en la memoria *cache* las palabras requeridas por el total de referencias a memoria. El tiempo efectivo de acceso se calcula dividiendo el tiempo total consumido en el acceso a memoria (sumando los tiempos de acceso de la memoria principal y la memoria *cache*) por el número total de referencias a memoria. Las ecuaciones correspondientes son las siguientes:

$$\text{Tasa de aciertos} = \frac{\text{N° de veces que la palabra buscada está en } \textit{cache}}{\text{N° total de accesos a memoria}}$$

$$\text{Tiempo efectivo de acceso} = \frac{(\text{N° aciertos})(\text{Tiempo por acierto}) + (\text{N° fallas})(\text{Tiempo por falla})}{\text{N° total de accesos a memoria}}$$

Considérese el cálculo de la tasa de aciertos y del tiempo efectivo de acceso de un programa que se ejecuta en una computadora en la que se dispone de una memoria *cache* de asignación directa, con cuatro líneas de 16 palabras. La distribución física de la memoria *cache* y de la memoria principal se muestran en la figura 7.17. El tiempo de acceso a la memoria *cache* es de 80 ns, en tanto que el tiempo requerido para transferir un bloque desde la memoria principal a la memoria *cache* es de 2.500 ns. Se supone que la arquitectura utiliza carga inmediata y que la memoria *cache* está vacía en el instante inicial. Se ejecuta un programa desde la posición de memoria 48 hasta la 95, tras lo cual el programa repite 10 veces un lazo entre las direcciones 15 y 31 antes de detenerse.

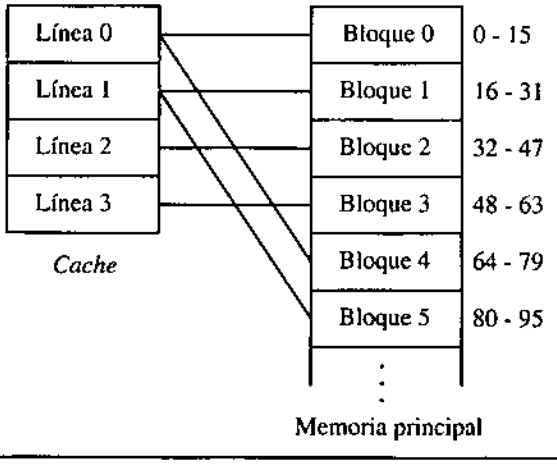


Figura 7.17 • Un ejemplo de una memoria *cache* de asignación directa.

En la figura 7.18 se registran los eventos que suceden durante la ejecución del programa. Dado que la memoria al comienzo se encuentra vacía, la primera instrucción a ejecutarse provoca una falla. Se produce entonces una falla en la dirección 48, lo que provoca la carga del bloque número 3 de memoria principal en la línea número 3 de la memoria *cache*. El primer acceso a memoria tarda 2.500 ns en completarse. Como se utiliza el procedimiento de carga inmediata, la palabra que provocó la falla en la dirección 48 se transfiere a la CPU mientras se carga el resto del bloque en la línea de memoria *cache*. El próximo evento contiene 15 aciertos, entre las direcciones 49 y 63. En forma similar, se registran los eventos que siguen, y se obtiene como resultado un total de 213 aciertos y 5 fallas. El número total de accesos es de 213 + 5 = 218. La tasa de aciertos y el tiempo efectivo de acceso se calculan en la forma siguiente:

$$\text{Tasa de aciertos} = \frac{213}{218} = 97{,}7\%$$

$$\text{Tiempo efectivo de acceso} = \frac{213 \cdot 80\ \text{ns} + 5 \cdot 2.500\ \text{ns}}{218} = 136\ \text{ns}$$

Si bien la tasa de aciertos es del 97,7%, el tiempo efectivo de acceso del modelo usado en el ejemplo es, aproximadamente, un 75% mayor que el tiempo de acceso propio de la memoria *cache*. Esto se debe al tiempo apreciablemente largo que se pierde cuando se tiene acceso a un bloque desde la memoria principal.

Evento	Posición	Tiempo	Comentario
1 Falla	48	2500ns	Bloque de memoria 3 a ranura de memoria *cache* 3
15 Aciertos	49-63	80ns×15=1200ns	
1 Falla	64	2500ns	Bloque de memoria 4 a ranura de memoria *cache* 0
15 Aciertos	65-79	80ns×15=1200ns	
1 Falla	80	2500ns	Bloque de memoria 5 a ranura de memoria *cache* 1
15 Aciertos	81-95	80ns×15=1200ns	
1 Falla	15	2500ns	Bloque de memoria 0 a ranura de memoria *cache* 0
1 Falla	16	2500ns	Bloque de memoria 1 a ranura de memoria *cache* 1
15 Aciertos	17-31	80ns×15=1200ns	
9 Aciertos	15	80ns×9=720ns	Últimas nueve iteraciones del lazo
144 Aciertos	16-31	80ns×144=11.520ns	
Total de aciertos = 213 Total de fallas = 5			

Figura 7.18 • Tabla de eventos para un programa que se ejecuta sobre una arquitectura con una memoria *cache* de asignación directa.

7.6.6 Memorias cache multinivel

A medida que fue creciendo el tamaño de los circuitos integrados de silicio, así como la densidad de integración de los elementos que lo componen, se hizo posible incluir la memoria *cache* en el mismo circuito integrado que el procesador. Dado que la velocidad de procesamiento interna del circuito integrado es mayor que la velocidad de comunicación entre circuitos integrados, una memoria *cache* integrada en el mismo circuito que el procesador puede ser más rápida que una memoria *cache* externa al circuito procesador. No obstante, la tecnología actual no ofrece la densidad suficiente como para colocar toda la memoria *cache* en el mismo circuito integrado del procesador. Por tal motivo, se han desarrollado **memorias *cache* de múltiples niveles** en las cuales el nivel más rápido, L1, se integra en el mismo circuito integrado que contiene al procesador, en tanto que el resto de la memoria se coloca fuera de dicho circuito integrado. Las memorias *cache* de datos e instrucciones se mantienen separadas en el nivel L1, en tanto que en el nivel L2 se implementan **unificadas**, lo que significa que la misma memoria contiene tanto datos como instrucciones.

Con el objeto de calcular la tasa de aciertos y el tiempo efectivo de acceso en una memoria *cache* multinivel, deben analizarse los aciertos y las fallas de ambas memorias. Las ecuaciones siguientes indican la tasa de aciertos general y el tiempo efectivo de acceso general para una memoria *cache* de dos niveles. H_1 es la tasa de aciertos de la memoria *cache* integrada al procesador, H_2 es la tasa de aciertos de la memoria externa y T_{EF} es el tiempo efectivo de acceso del conjunto. El método puede extenderse a cualquier número de niveles.

$$H_1 = \frac{\text{(N° de veces en que la palabra buscada está en la memoria } \textit{cache} \text{ integrada)}}{\text{(N° total de accesos a memoria)}}$$

$$H_2 = \frac{\text{(N° de veces en que la palabra buscada está en la memoria } \textit{cache} \text{ externa)}}{\text{(N° de veces en que la palabra buscada no está en la memoria integrada)}}$$

T_{EF} = [(N° de aciertos de memoria integrada)(Tiempo de acierto de la memoria integrada) + (N° de aciertos de la memoria externa)(Tiempo de acierto de la memoria externa) + (N° de fallas de la memoria externa) (Tiempo de falla de la memoria externa)] / Número total de accesos a memoria

7.6.7 Administración de la memoria cache

La administración de la memoria *cache* presenta un problema complejo para el programador de sistemas. Si una posición de memoria determinada representa un elemento de entrada-salida, como puede ocurrir en los sistemas de entrada-salida mapeada en memoria, no debería aparecer en la memoria *cache* bajo ningún punto de vista. Si se la coloca en la memoria *cache*, el valor en el elemento de entrada-salida puede cambiar, y ese cambio puede no verse reflejado en el valor del dato almacenado en la memoria *cache*. Esto se conoce como "información rancia": la copia de los datos almacenada en la memoria *cache* es vieja comparada con el valor presente en la memoria principal. De la misma forma, en ambientes **multiprocesadores de memoria compartida** (véase el capítulo 10), en los que más de un procesador puede acceder a la misma memoria principal, tanto el valor almacenado en la memoria *cache* como el valor almacenado en la memoria principal pueden volverse rancios debido a la actividad de uno o más de uno de los procesadores del sistema. Como mínimo, la memoria *cache* de un ambiente multiprocesador debería implementar una política de escritura inmediata para aquellas líneas de la memoria *cache* asignadas a posiciones de memoria compartidas.

Por estas razones, y por otras, la mayoría de las arquitecturas modernas admiten que el programador del sistema tenga algún grado de control sobre la memoria *cache*. Por ejemplo, la memoria *cache* del procesador Motorola PPC 601, que normalmente utiliza una política de escritura diferida (*write back*), puede usarse con el formato de escritura inmediata sobre determinadas líneas. Otras instrucciones permiten especificar líneas como no almacenables en memoria *cache*, o permiten marcarlas como inválidas, como cargadas o como eliminadas.

Internamente a la memoria *cache*, las políticas de reemplazo (para memorias asociativas y asociativas por conjuntos) necesitan ser implementadas eficientemente. Una implementación eficiente del algoritmo de reemplazo **LRU** se logra con el **Algoritmo Neat Little** (origen desconocido). Siguiendo con el ejemplo de la memoria *cache* utilizada en la sección 7.6.5, se construye una matriz en la que cada línea de la memoria *cache* se representa por una fila y una columna, como lo muestra la figura 7.19. El contenido inicial de todas las celdas es 0. Cada vez que se accede a una línea, se escribe un 1 en cada celda de la fila de la tabla que corresponde a esa línea. Luego se completan con ceros todas las celdas de la columna correspondiente a esa línea. Cada vez que se requiere una línea, la fila que solo contiene ceros es la más antigua y, por consiguiente, es la próxima en ser utilizada. Al comienzo del proceso, hay más de una fila que contiene solo ceros, por lo que hará falta diseñar un mecanismo de desempate. En este caso, un método que funciona, que se utilizará en el resto del ejemplo, es el que elige la primera fila que tenga todas las celdas en 0.

El ejemplo de la figura 7.19 muestra la configuración de la matriz a medida que los bloques se acceden en el orden 0, 2, 3, 1, 5, 4. Inicialmente, la matriz se completa con ceros. Luego de una referencia al bloque 0, la fila correspondiente a dicho bloque se llena con unos y la columna correspondiente se llena con ceros. En este ejemplo, el bloque 0 se coloca en la línea 0, pero en otros casos, podría aparecer en cualquier otra ubicación. El proceso sigue hasta que, al final de la secuencia 0, 2, 3, 1, se han utilizado todas las líneas de la memoria *cache*. Con el objeto de ingresar el bloque siguiente (bloque 5) a la memoria *cache*, deberá liberarse primero alguna de las líneas. La fila correspondiente a la línea 0 solo contiene ceros, por lo que esta línea es la menos recientemente usada. El bloque 5 se ingresa en la línea 0. En forma similar, cuando se carga en la memoria *cache* el bloque 4, la línea que se escribe es la número 2.

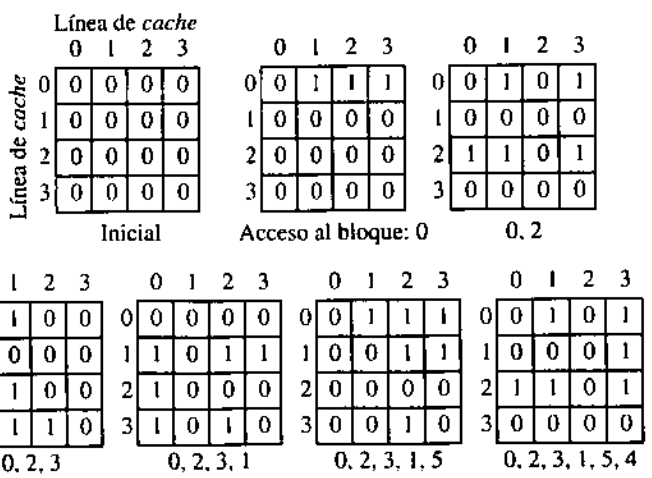


Figura 7.19 • Secuencia de funcionamiento del algoritmo Neat Little de reemplazo LRU para una memoria *cache* con cuatro líneas. El acceso a los bloques de memoria principal se realiza en la secuencia 0, 2, 3, 1, 5, 4.

7.7 Memoria virtual

A pesar de los enormes avances en la creación de memorias cada vez mayores en espacios cada vez menores, la memoria de una computadora parece ser algo así como el espacio de un armario, ya que por muy grande que sea nunca es suficiente. Un método económico para expandir el tamaño de la memoria principal es aumentar dicho tamaño con espacio en disco, lo que define uno de los aspectos de la **memoria virtual** que se analiza en esta sección. El almacenamiento en disco aparece muy cerca del fondo de la estructura jerárquica de memoria, con un costo por bit menor que el de la memoria principal; por este motivo, es razonable usar almacenamiento en disco para contener aquellas porciones de un programa o de una sección de datos que no cabe íntegramente en la memoria principal. En un aspecto diferente de la memoria virtual, el uso de la memoria logra gran flexibilidad con el manejo de esquemas complejos de asignación de memoria. Estos dos aspectos de la memoria virtual se analizan a continuación.

7.7.1 Superposiciones (overlays)

Una primera aproximación a la utilización de almacenamiento en disco para aumentar el tamaño de la memoria principal hace uso de **superposiciones**. En las superposiciones, un programa en ejecución sobrescribe su propio código con otro código, en la medida de lo necesario. En este escenario, el programador tiene la responsabilidad de administrar el uso de la memoria. La figura 7.20 muestra un programa que contiene una rutina principal y tres subrutinas *A*, *B* y *C*. La memoria física es menor que el tamaño del programa pero mayor que cualquier rutina individual. Una estrategia que permite manejar la memoria usando superposiciones consiste en modificar el programa para que pueda llevar el control de cuáles son las subrutinas que están en memoria y para que pueda leer el código de la subrutina a medida que la requiera. En general, la rutina principal sirve como elemento de **control** (*driver*) del sistema en su conjunto. El controlador permanece residente en memoria mientras las demás rutinas se cargan y se descargan a medida que se hace necesario.

La figura 7.20 muestra un **gráfico de particiones** creado para el programa del ejemplo. El gráfico de particiones identifica cuáles de las rutinas pueden superponerse a otras sobre la base de los distintos llamados entre las subrutinas. Para este ejemplo, la rutina principal se encuentra siempre presente y supervisa al subconjunto de subrutinas que se encuentra en memoria. Las subrutinas *B* y *C* se mantienen en la misma partición porque *B* invoca a *C*, pero la subrutina *A* tiene una partición propia porque solo la invoca el programa principal. La partición 0 puede superponerse entonces sobre la partición 1, en tanto que la partición 1 puede superponerse sobre la partición 0.

Si bien este método funcionará correctamente en una variedad de situaciones, una solución más prolija sería la de permitir que el sistema operativo administre las particiones. No obstante, cuando se carga más de un programa en memoria, las rutinas que manejan las superposiciones no pueden funcionar sin interacturar con el sistema operativo para

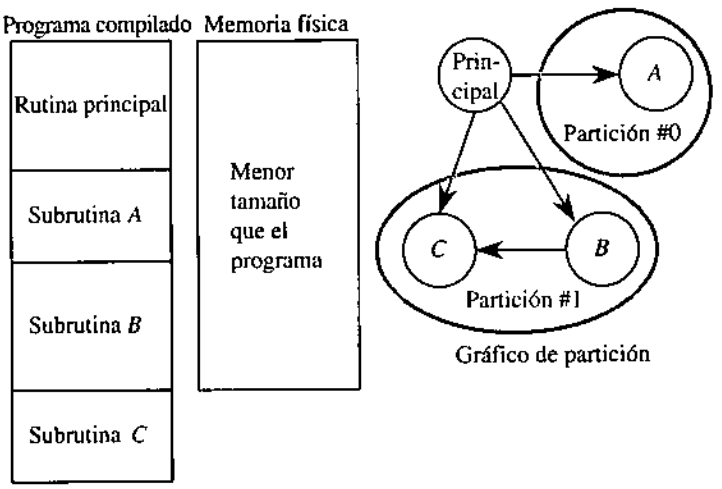


Figura 7.20 • Gráfico de particiones de un programa con una rutina principal y tres subrutinas.

determinar cuáles son las porciones de memoria disponibles. Este escenario introduce un elevado nivel de complejidad en el manejo del proceso de superposición, debido a que ocasiona una fuerte interacción entre el sistema operativo y cada programa. El **método de paginación**, que se describe en la sección siguiente, es una alternativa, ya que puede ser manejado por el sistema operativo.

7.7.2 Paginación

El proceso de paginación es una forma de superposición automática administrada por el sistema operativo. El espacio de direcciones se divide en bloques de igual tamaño, llamados **páginas**. Las páginas tienen normalmente un tamaño que es potencia de dos, como, por ejemplo, $2^{10} = 1.024$ bytes. El proceso de paginación hace que la memoria física parezca mayor de lo que es en realidad, por medio de la asignación del espacio de direcciones de la memoria física a alguna porción del espacio de direcciones de la memoria virtual, el que normalmente se encuentra almacenado en un disco. La figura 7.21 ilustra un esquema de asignación de memoria virtual, donde ocho páginas virtuales son asignadas a cuatro **marcos de página**.

La implementación de la memoria virtual debe manejar referencias que se realizan fuera de la porción de espacio virtual asignada al espacio físico. La siguiente secuencia de eventos ocurre generalmente cuando se hace referencia a una posición de memoria virtual que no está en la memoria física, lo que se denomina **falla de página**.

1. Se identifica uno de los marcos para su sobreescritura. El contenido del marco se escribe en memoria secundaria si hubiese habido cambios, para poder rescatar los cambios antes de sobrescribir el marco.

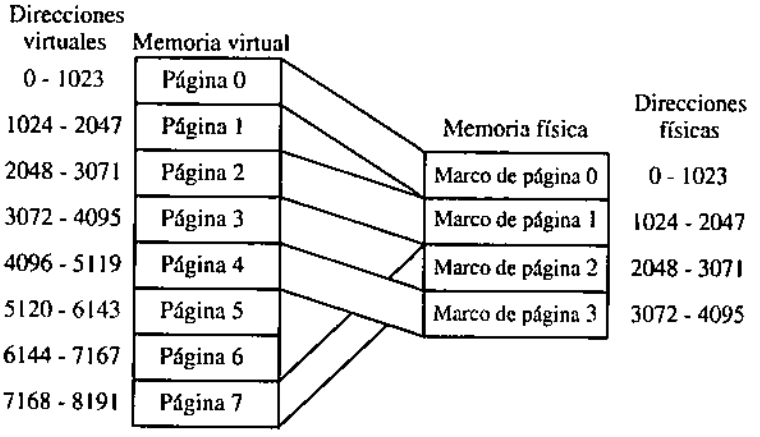


Figura 7.21 • Esquema de asignación entre una memoria virtual y una memoria física.

2. Se ubica en la memoria secundaria la página virtual a la que se desea acceder, la que se escribe en memoria física, en el marco determinado en el punto 1.
3. Se actualiza la tabla de páginas (véase la figura 7.22) para asignar la nueva sección de memoria virtual a la memoria física.
4. Se continúa con la ejecución.

Para la memoria virtual de la figura 7.21, existen $2^{13} = 8192$ posiciones virtuales, por lo que el programa en ejecución debe generar direcciones de 13 bits, las que se interpretan como un número de página de 3 bits y un desplazamiento, dentro de la página, de 10 bits. Dado el número de página de 3 bits, se debe determinar si la página se encuentra en uno de los cuatro marcos o si se encuentra en memoria secundaria. Con el objeto de llevar el control de las páginas que están en la memoria física, se debe administrar una **tabla de páginas**, como la que se ilustra en la figura 7.22, y que corresponde a la asignación de la figura 7.21.

La tabla de páginas tiene tantas entradas como páginas virtuales existen. El bit de presencia indica si la página correspondiente está o no en memoria física. El campo de dirección de disco es un puntero a la posición en la que puede encontrarse la página correspondiente en una unidad de disco. El sistema operativo normalmente administra los accesos a disco, por lo tanto, la tabla de páginas solo requiere mantener las direcciones de disco que el sistema operativo asigna a los bloques cuando el sistema arranca. Las direcciones de disco normalmente no cambian durante el curso del cálculo. El campo de marco de página indica cuál de los marcos de página física contiene una página virtual, si la página se encuentra en memoria física. Los campos de marco de página son inválidos para aquellas páginas que no se encuentran en memoria, y que se indican con "xx" en la figura 7.22.

Con el objeto de traducir una dirección virtual en una dirección física, se toman los dos bits del marco de la tabla de páginas y se agregan a la izquierda del desplazamiento de 10 bits, lo que genera la dirección física de la palabra referida. Considérese la situación de la figura 7.23, en la que se hace referencia a la dirección virtual 1 0011 0100 0101. Los tres

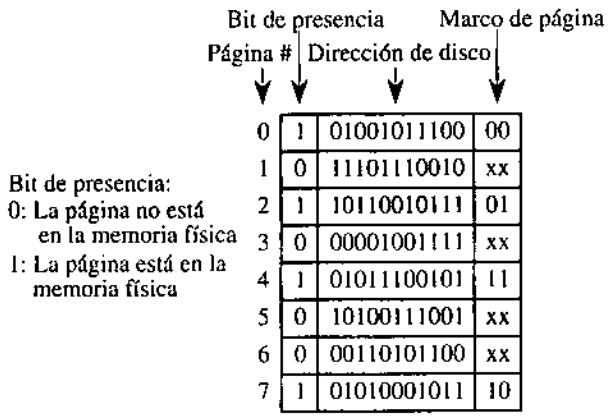


Figura 7.22 • Tabla de páginas para una memoria virtual.

bits más significativos de la dirección virtual (100) identifican la página. La palabra binaria que aparece en el campo de marcos de página (11) se agrega a la izquierda del desplazamiento de 10 bits (11 0100 0101), con lo que la dirección resultante (1111 0100 0101) indica cuál es la dirección de memoria física que contiene la palabra referida.

El tiempo que requiere la carga de un programa en memoria puede ser relativamente largo. De hecho, puede ocurrir que el programa entero no se ejecute nunca, por lo tanto, el tiempo requerido para cargar el programa desde un disco a memoria puede reducirse si se carga solo la porción del programa que se requiere en un determinado intervalo de tiempo. El esquema de **paginación por demanda** no carga ninguna página en memoria hasta que no se produzca una falla de paginación. Luego de tener un programa ejecutándose durante un cierto tiempo, la memoria física contiene solo aquellas páginas que están siendo usadas (el **conjunto de trabajo**), por lo que la paginación por demanda no tiene un impacto significativo sobre programas que funcionan durante largo tiempo.

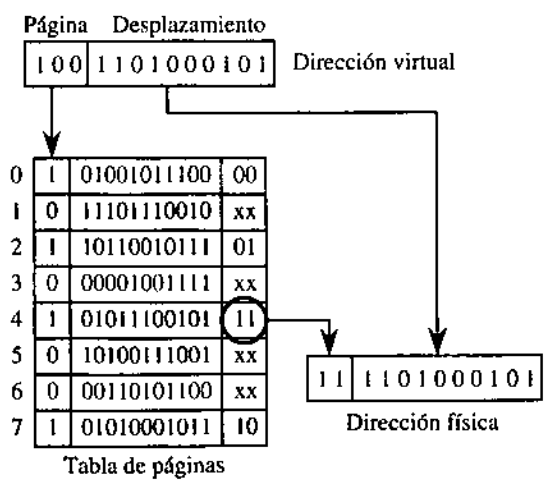


Figura 7.23 • Traducción de una dirección virtual hacia una dirección física.

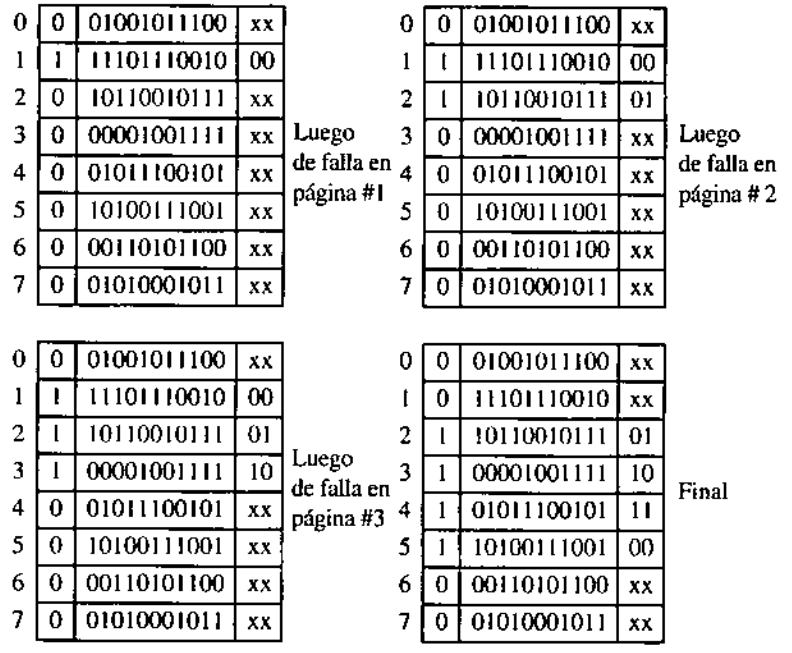


Figura 7.24 • La configuración de la tabla de páginas cambia con la ejecución de un programa. Inicialmente, la tabla está vacía. En la configuración final, hay cuatro páginas en la memoria física.

Considérese nuevamente la asignación de memoria de la figura 7.21. El tamaño del espacio de direcciones virtuales es de 2^{13} palabras, en tanto que el espacio de direcciones físicas es de 2^{12} palabras. Existen ocho páginas de 2^{10} palabras cada una. Se supone que la memoria se encuentra vacía en su estado inicial y que se utiliza paginación por demanda para un programa que se ejecuta entre las direcciones 1030 y 5300. La secuencia de ejecución accederá a las páginas 1, 2, 3, 4 y 5, en ese orden. La política de reemplazo de páginas es FIFO (*first in first out*, primero que entra, primero que sale). La figura 7.24 ilustra la configuración de la tabla de páginas a medida que se avanza en la ejecución. El primer acceso a memoria generará una falla de página en la dirección virtual 1030, que se encuentra en la página 1. La página se trae hacia la memoria física, y se actualiza la tabla de páginas con el bit de validez y el campo de marco de páginas correspondientes. Se continua la ejecución, hasta que se requiere cargar la página 5, la que fuerza la salida de la página 1 debido a la política de reemplazo utilizada. La configuración final de la tabla de páginas que se muestra en la figura 7.24 es la que se obtiene luego del acceso a la posición 5300.

7.7.3 Segmentación

De acuerdo con el análisis realizado hasta el momento, la memoria virtual es unidimensional, entendiéndose con este concepto que las direcciones pueden crecer hacia arriba o hacia abajo. La **segmentación** divide el espacio de direcciones en segmentos que pueden ser de tamaño arbitrario. Cada **segmento** tiene su propio espacio unidimensional de di-

recciones. Esto permite mantener tablas, pilas y otras estructuras de datos como entidades lógicas que crecen sin entrar en conflicto unas con otras. La segmentación admite **esquemas de protección,** de modo tal que un segmento determinado pueda ser definido como "de lectura solamente" para evitar cambios, o "de ejecución solamente" para evitar que sea copiado sin autorización. Este esquema también impide que los usuarios intenten escribir datos en zonas de programa.

Cuando se utiliza la segmentación en la memoria virtual, el tamaño del espacio físico de direcciones de cada segmento puede ser muy grande, por lo que la memoria física a dedicar a cada segmento no se define hasta que no se la necesita.

La figura 7.25 ilustra una memoria segmentada. El código ejecutable de un programa procesador de texto se encuentra cargado en el segmento 0. Este segmento se marca como "de ejecución solamente" y se protege contra escrituras. En el segmento 1 se encuentra el espacio de datos del usuario 0, por lo que se lo marca como de lectura y escritura exclusivo para el usuario 0, de modo tal que ningún otro usuario pueda tener acceso al área. El segmento 2 se utiliza para el espacio de datos del usuario 1, por lo que se lo identifica como de lectura y escritura para el usuario 1. Ambos usuarios, 0 y 1, pueden utilizar el mismo procesador de texto, caso en el cual el código del segmento 0 se comparte, pero cada usuario tiene su propio espacio de datos.

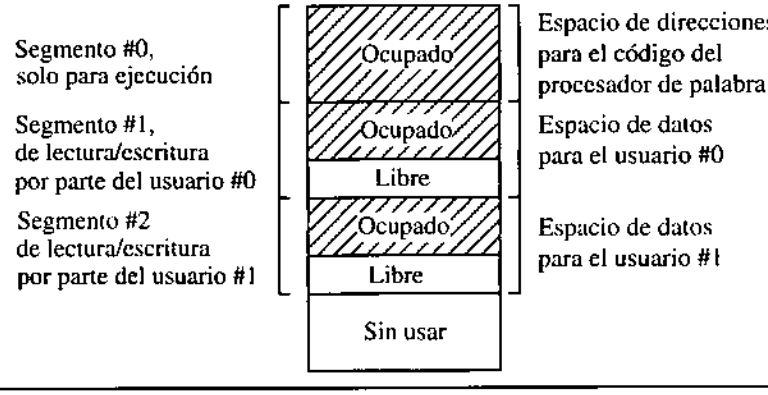


Figura 7.25 • Una memoria segmentada permite que dos usuarios compartan el mismo procesador de texto.

Segmentación no es lo mismo que paginación. Con el procedimiento de paginación, el usuario no ve la superposición automática. Con la segmentación, el usuario conoce dónde se encuentran los límites del segmento. El sistema operativo administra la protección y la asignación, en consecuencia, el usuario común no necesita tratar con los aspectos administrativos. No obstante, un usuario más sofisticado, como un programador de computadoras, puede encontrarse frecuentemente con el tema, tal como ocurre cuando un puntero a un arreglo intenta superar los límites de un segmento por culpa de un programa errante.

Con el objeto de determinar una dirección en una memoria segmentada, el programa usuario debe especificar un número de segmento y una dirección dentro del mismo. El sistema operativo traduce la dirección segmentada del usuario en una dirección física.

7.7.4 Fragmentación

Cuando se enciende una computadora, pasa por una secuencia de inicialización que carga el sistema operativo en memoria. Una porción del espacio de direcciones puede estar reservada para los dispositivos de entrada-salida, por lo que el resto queda disponible para ser usado por el sistema operativo. Esta porción remanente del espacio de direcciones puede estar solo parcialmente completa con memoria física: el resto comprende una zona muerta, a la que nunca podrá accederse por cuanto no hay electrónica que responda a las direcciones de esa zona muerta.

La figura 7.26a ilustra el estado de una memoria inmediatamente antes de la secuencia de inicialización. El "área libre" es la zona de memoria de que dispone el sistema operativo para la carga y ejecución de programas. Durante el curso de la operación, se cargan y se ejecutan programas de distintos tamaños. Cuando un programa finaliza su ejecución, el espacio de memoria asignado al mismo se devuelve al sistema operativo. A medida que se produce la carga y ejecución de los programas, el área libre se va subdividiendo en un conjunto de pequeñas áreas, ninguna de las cuales tiene el tamaño suficiente como para contener un programa, salvo que algunas o todas las áreas libres se combinen en un solo sector de tamaño mayor. Este problema, conocido como **fragmentación**, se contrapone con el criterio de la segmentación, dado que los segmentos deben aparecer asignados dentro de un único espacio lineal de direcciones.

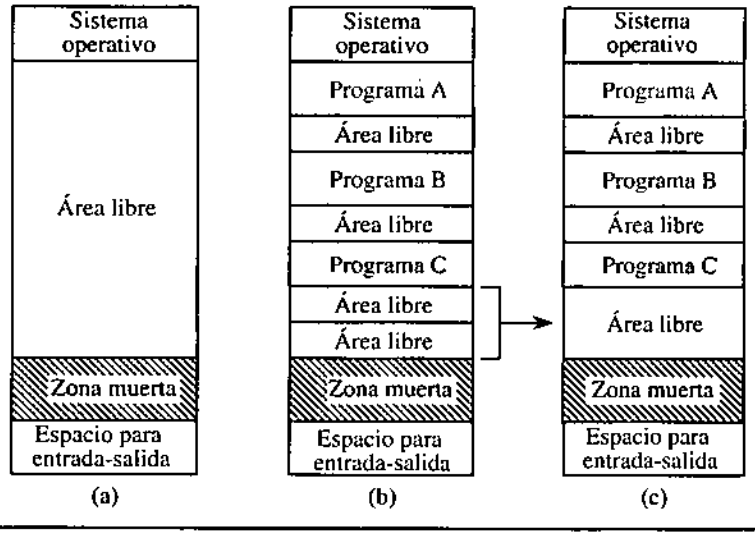


Figura 7.26 • (a) Área libre de memoria luego de la inicialización, (b) luego de la fragmentación, (c) luego de la fusión de áreas libres.

La figura 7.26b ilustra el problema de la fragmentación. Cuando el sistema operativo necesita encontrar un área libre lo suficientemente grande como para contener un programa, es muy poco probable que encuentre algo de tamaño exacto. En general, el área libre

será mayor que el tamaño del programa, lo que produce como efecto la subdivisión de las áreas libres en áreas más pequeñas, en la medida que los tamaños de los programas no coincidan con los tamaños de las áreas libres. Un método de asignación de programas a esas áreas libres consiste en la técnica del **primer lugar libre** (*first fit*), en el cual se analizan las áreas libres hasta que aparezca una cuyo tamaño sea suficiente como para satisfacer las necesidades del programa. Otro método es el del **mejor lugar libre** (*best fit*), en el cual se utiliza el área libre que desaproveche la menor cantidad de espacio. Este segundo método aprovecha mejor el espacio de memoria que el anterior, pero tiene como desventaja el mayor tiempo requerido para revisar todas las áreas libres.

Independientemente del algoritmo utilizado, el proceso de asignación de programas o datos en áreas libres tiende a producir espacios libres menores (véase la obra de D. Knuth). Esto hace difícil poder encontrar un área libre de direcciones contiguas lo suficientemente grande como para satisfacer las necesidades del sistema operativo. Una aproximación a la solución del problema unifica áreas libres adyacentes en un único espacio libre de mayor tamaño. En el ejemplo de la figura 7.26b, las dos áreas libres adyacentes se combinan en un único espacio libre, como lo muestra la figura 7.26c.

7.7.5 Memoria virtual versus memoria cache

La memoria virtual se divide en páginas; estas son relativamente grandes cuando se las compara con los bloques de memoria *cache*, los que no suelen tener un tamaño mayor a algunas palabras. La memoria *cache* mantiene copias de los bloques más fuertemente usados, los que también se encuentran en memoria principal y en la imagen de memoria virtual que se almacena en un disco magnético. Cuando en una computadora que tiene tanto memoria *cache* como memoria virtual se hace referencia a memoria, la electrónica de la memoria *cache* encuentra dicha referencia primero y satisface el requerimiento si la palabra está efectivamente en la memoria *cache*. Si la palabra requerida no está en la memoria *cache*, el bloque que la contiene se transfiere desde la memoria principal hacia la memoria *cache* y se utiliza la palabra de referencia desde la memoria *cache*. Si la página que contiene la palabra no está en la memoria principal, se la carga en memoria principal desde una unidad de disco, tras lo cual se transfiere el bloque que contiene la palabra a la memoria *cache* para permitir satisfacer la referencia.

El uso de memoria virtual puede provocar algunas interacciones intrincadas con la memoria *cache*. Por ejemplo, dado que en un mismo momento puede haber más de un programa utilizando la memoria *cache* y la memoria virtual, las estadísticas de tiempos de ejecución para dos procesos de un mismo programa ejecutándose sobre el mismo conjunto de datos pueden dar resultados diferentes. Asimismo, cuando se hace necesario reescribir a memoria principal un bloque sucio, es posible que el marco de página que contenía la página virtual correspondiente ya haya sido eliminado. Esto obliga a cargar nuevamente la página en la memoria principal desde la memoria secundaria, con el objeto de limpiar el bloque sucio desde la memoria *cache* hacia la memoria principal.

7.7.6 El buffer de traducción anticipada (Translation Lookaside Buffer - TLB)

El mecanismo de memoria virtual es una solución elegante al problema del acceso a programas y archivos de datos de gran tamaño, pero trae asociado un inconveniente significativo. Para acceder a un valor almacenado en memoria se requieren, al menos, dos referencias a memoria. Una referencia corresponde a la tabla de páginas para encontrar el marco de página física, en tanto que la otra referencia corresponde al valor real del dato. Una solución a este problema se logra a través de un **buffer de traducción anticipada** (TLB, *translation lookaside buffer*).

El TLB consiste en una memoria asociativa pequeña ubicada generalmente dentro de la CPU, la que almacena las traducciones más recientes realizadas desde la memoria virtual a las direcciones físicas. La primera vez que se traduce una determinada dirección virtual hacia una dirección física, esta traducción se almacena en el TLB. Cada vez que la CPU genera una dirección virtual, se analiza el contenido del TLB para ver si contiene la dirección generada. Si el número de página virtual existe, el TLB devuelve el número de página física, el que puede ser inmediatamente enviado a la memoria principal (aun cuando, normalmente, la memoria *cache* capturaría la referencia a memoria principal satisfaciendo el requerimiento desde la misma).

La figura 7.27 ilustra un modelo de *buffer* de traducción anticipada. El elemento contiene 8 entradas de un sistema de 32 páginas y 16 marcos de página. El campo de página virtual tiene 5 bits porque hay $2^5 = 32$ páginas. Asimismo, el campo de página física tiene 4 bits debido a que hay $2^4 = 16$ marcos de página.

Validez	Número de página virtual	Número de página física
1	01001	1100
1	10111	1001
0	-----	----
0	-----	----
1	01110	0000
0	-----	----
1	00110	0111
0	-----	----

Figura 7.27 • Un *buffer* de traducción anticipada que contiene 8 entradas de un sistema de 32 páginas virtuales y 16 marcos de página.

Las fallas del TLB se manejan en forma similar a las demás fallas de memoria. Cuando se produce una falla en el *buffer* de traducción, se envía la dirección virtual al sistema de memoria virtual, donde se la busca en la tabla de páginas de la memoria principal. Si se la encuentra en la tabla de páginas, se actualiza el TLB, para que una próxima referencia a esa página no se convierta en una nueva falla de TLB.

7.8 Temas avanzados

Esta sección analiza dos temas de importancia práctica en el diseño de sistemas de memoria: los árboles decodificadores y las memorias direccionables por contenido. Los primeros se hacen necesarios en memorias de gran tamaño, las últimas, en memorias *cache* asociativas, por ejemplo un TLB, o en otras situaciones en las que deban realizarse búsquedas de datos en función de su valor más que en función del lugar en el que se encuentran almacenados.

7.8.1 Árboles decodificadores

Los circuitos decodificadores (véase el apéndice A) no resultan prácticos cuando se requieren grandes tamaños debido a limitaciones prácticas en la cantidad de entradas y salidas. El circuito decodificador de la figura 7.28 ilustra el problema. Para N bits de direcciones, cada compuerta Y requiere N entradas. Cada línea de dirección debe enviarse a 2^N compuertas Y. La profundidad del circuito es de dos niveles de compuertas.

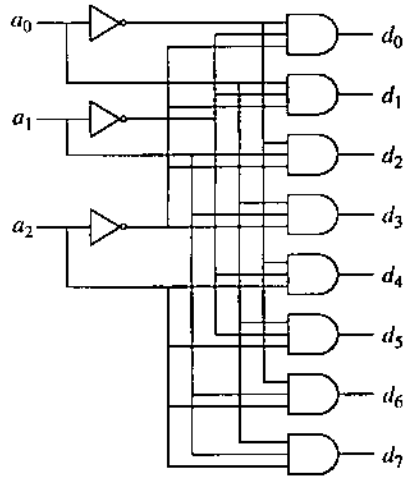


Figura 7.28 • Un decodificador convencional.

El circuito que se muestra en la figura 7.29a representa un **árbol decodificador**, que reduce la cantidad de entradas y salidas por medio del aumento del número de niveles lógicos del circuito. En este caso, cada compuerta Y tiene F entradas (en este ejemplo, F = 2), y solo la línea de direcciones que se ingresa en el nivel más alejado (a_0 en este caso) se envía a $2^N/2$ compuertas Y. El número de niveles ha sido incrementado a $log_F(2^N)$. El gran requerimiento de salidas en los bits de direcciones menos significativos puede ser un problema, pero se soluciona en forma sencilla, sin incrementar la cantidad de niveles del circuito, por medido del agregado de amplificadores de corriente (*buffers*) en los primeros niveles, según se muestra en la figura 7.29b.

Por consiguiente, la cantidad de niveles de un árbol decodificador de direcciones de memoria es de $log_F(2^N)$, la cantidad de líneas de entrada es 2^N, y la cantidad máxima de entradas y salidas a cada compuerta lógica dentro del decodificador es F.

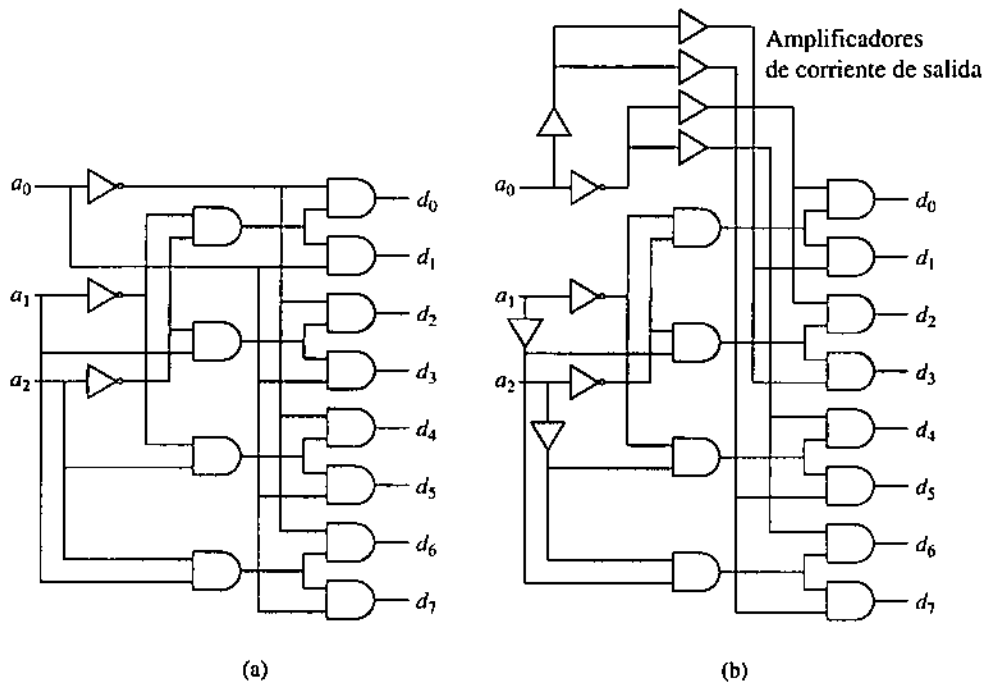


Figura 7.29 • (a) Un árbol decodificador. (b) Un árbol decodificador con compuertas de dos entradas y dos cargas en las salidas.

7.8.2 Decodificadores para memorias de acceso aleatorio muy grandes

Cuando se trabaja con memorias de acceso aleatorio de gran tamaño, y si no se emplea el esquema de decodificación 2-1/2D, se utilizan árboles decodificadores con el objeto de mantener dentro de límites razonables la cantidad de entradas y de salidas. En una memoria de acceso aleatorio se identifica cada posición de memoria que pertenece a un espacio de 2^M posiciones con una única palabra de M bits. Con el objeto de acceder a una posición en particular, se presenta la palabra de direcciones en la entrada de un árbol decodificador de M niveles y 2^M hojas. Comenzando desde la base del árbol (el nivel superior del mismo), en cada nivel de orden i se toma una determinación referida al bit i de la palabra de dirección. Si en el nivel i el bit es 0, se circula por el árbol hacia la izquierda; en caso contrario, se avanza hacia la derecha. La hoja de destino está en el nivel $M - 1$ (contando desde 0) y existe una sola hoja por cada dirección de memoria.

La estructura en forma de árbol da por resultado un tiempo de acceso que es logarítmico con respecto al tamaño de la memoria. Esto es, si la memoria contiene N palabras, el

tiempo de acceso de la memoria resulta ser $O[log_F N]$, donde F es el *fan out* de las compuertas lógicas del árbol. (Aquí se supone nuevamente que se trabaja con un *fan out* de dos.) Para una memoria de tamaño N, se requieren $M = [log_F N]$ bits de dirección para identificar unívocamente cada palabra. A medida que crece el número de palabras en la memoria, la longitud de la palabra de direcciones crece en forma logarítmica, de modo tal que se agrega un nivel de profundidad al árbol decodificador cada vez que el tamaño de la memoria se duplica. Como ejemplo práctico, considérese una memoria de 128 Mpalabras, que requiere 27 niveles de decodificación (dado que 2^{27} = 128 Mpalabras). Si se supone que las compuertas lógicas del árbol decodificador conmutan en 2 ns, puede accederse a cualquier dirección en 54 ns.

La figura 7.30 muestra un árbol decodificador de cuatro niveles para una memoria de acceso aleatorio de 16 palabras. Como ejemplo de funcionamiento del árbol, supóngase que la dirección a seleccionar es 1011. El bit más significativo es 1, por lo que se circula hacia la derecha del nivel 0, como lo indica la flecha. El siguiente bit más significativo es un 0, por lo que desde el nivel 1 se circula hacia la izquierda; el bit siguiente es un 1, con lo que se vuelve a circular hacia la derecha, lo mismo que en el caso del bit menos significativo, lo que permite encontrar el elemento direccionado a la altura del nivel 3.

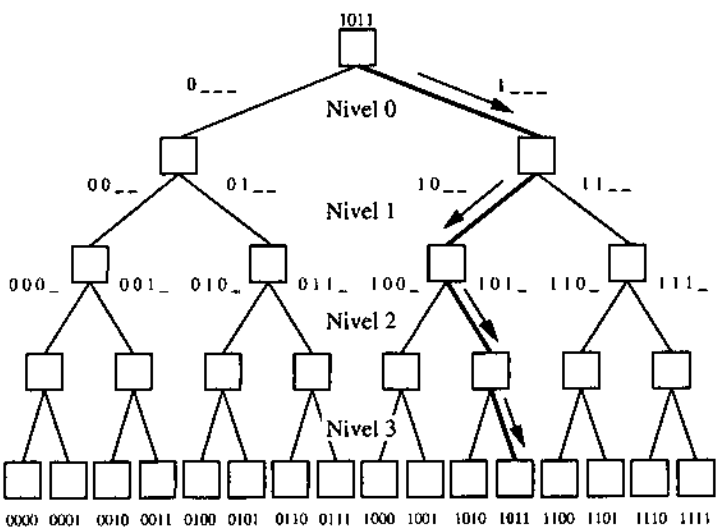


Figura 7.30 • Un árbol decodificador para una memoria de acceso aleatorio de 16 palabras.

7.8.3 Memorias de direccionamiento por contenido (asociativas)

En una memoria de acceso aleatorio convencional, se aplica una dirección a la memoria, con lo que se permite el acceso a la posición dada para su lectura o escritura. En una **memoria direccionable por contenido** (CAM, *contents addressable memory*), conocida también como memoria asociativa, se aplica a la entrada de la memoria una palabra for-

mada por **campos**, recibiéndose como respuesta la dirección (o índice) resultante en el caso en que la palabra o el campo estuviesen presentes en la memoria. La ubicación física de una palabra almacenada en una memoria asociativa, generalmente, no es tan significativa como los valores contenidos en los campos de la palabra. La figura 7.31 indica las relaciones existentes entre direcciones, valores y campos de memorias de acceso aleatorio y de memorias asociativas.

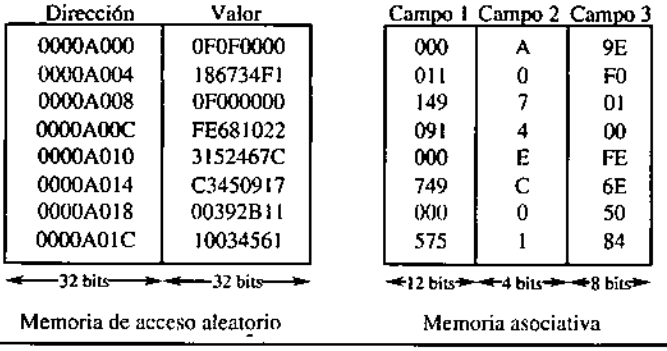


Figura 7.31 • Relación entre una memoria de acceso aleatorio y una memoria asociativa.

En una memoria de acceso aleatorio, los valores se almacenan en posiciones secuenciales y la dirección actúa como clave para ubicar la palabra. En este ejemplo se utilizan incrementos de dirección de cuatro bytes, dado que el tamaño de la palabra de datos es de cuatro bytes. En una memoria asociativa, los valores se almacenan en campos y, en principio, cualquier campo de una palabra puede usarse como clave para el resto de la palabra. Se pueden reordenar las palabras de la memoria asociativa sin que los contenidos de la misma sufran ningún cambio, dado que la posición física de los datos no tiene influencia sobre la interpretación de los campos. El reordenamiento de los contenidos de una memoria de acceso aleatorio puede cambiar íntegramente los significados de sus datos. Esta comparación sugiere que la memoria asociativa puede ser un elemento adecuado para almacenar información cuando el costo de mantener ordenados los datos es significativo.

Cuando se realiza la búsqueda de un determinado valor en una memoria de acceso aleatorio que no esté ordenada, se debe buscar en la totalidad de la memoria, palabra por palabra. Cuando el contenido de la memoria se mantiene en un cierto orden, sigue siendo necesario un cierto número de accesos con el objeto de encontrar el valor buscado o para determinar que el mismo no se encuentra almacenado en la memoria. En una memoria de direccionamiento por contenido, el valor buscado se emite a todas las palabras simultáneamente, y la lógica existente en el ámbito de cada palabra realiza una operación a nivel de campos para verificar si existe correspondencia. Así, el resultado de la búsqueda se determina en algunos pocos pasos. Pueden requerirse algunos pasos adicionales para recoger los resultados, pero, en general, el tiempo requerido para realizar una búsqueda en una memoria asociativa es menor que el de la memoria de acceso aleatorio en la misma tecnología.

A excepción de las aplicaciones de mantenimiento de etiquetas en memorias *cache* y de traducción de direcciones en aplicaciones de encaminamiento (véase el capítulo 8), las memorias de acceso asociativo no se usan en gran escala debido a la dificultad que implica la implementación de un diseño eficiente con tecnologías convencionales. Si se considera el diagrama en bloques de una memoria asociativa, como el de la figura 7.32, se incluye una unidad central de control que envía una señal de comparación a cada una de las 4096 celdas, en las que se realizan las comparaciones. El resultado se coloca en los bits de etiqueta T_i, los que se reúnen en un dispositivo de recolección de datos y se envían a la unidad de control central (nótese que la etiqueta, en este ejemplo, se utiliza en forma diferente a la forma en que se usa en la memoria *cache*). Cuando la unidad central de control carga el valor a buscar en el registro comparador, coloca una máscara para bloquear aquellos valores que no son parte del valor. Un pequeño procesador local ubicado en cada celda realiza una comparación entre su palabra local y el valor enviado, e informa el resultado de la comparación al dispositivo recolector.

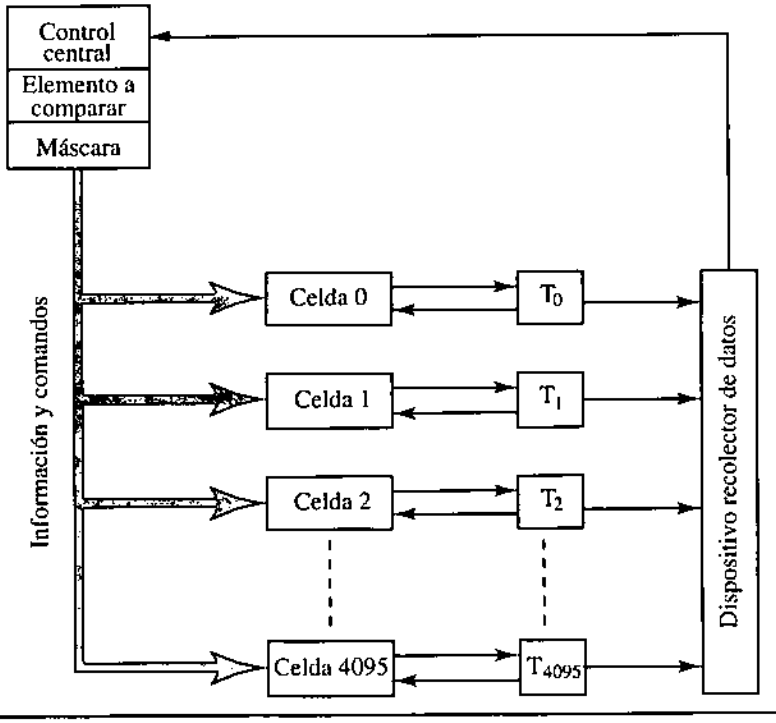


Figura 7.32 • Revisión de una memoria asociativa (adaptado de Foster).

Cuando se intenta implementar esta arquitectura asociativa en una tecnología convencional como la de **integración en muy alta escala** (VLSI) surgen una serie de problemas. La función de transmisión que envía el valor a comparar a todas las celdas puede implementarse con un tiempo de retardo pequeño si se utiliza una estructura de árbol. Se puede utilizar una estructura de árbol H (véase la obra de C. Mead y L. Conway) para confi-

gurar dicho árbol si se lo va a colocar en un único circuito integrado. Si el árbol no puede contenerse en un único circuito integrado, deben realizarse conexiones entre distintos circuitos integrados, lo que limita muy rápidamente la densidad circuital. Por ejemplo, un nodo de un árbol que tiene una única entrada de cuatro bits y dos salidas de cuatro bits necesita 12 terminales de entrada-salida y tres terminales de control si se ubica un solo nodo en el circuito integrado. Un subárbol con tres nodos requiere 25 terminales, en tanto que un subárbol de siete nodos requiere 45 terminales, de acuerdo con lo que se ilustra en la figura 7.33. Un subárbol de 63 nodos requiere 325 terminales, excluyendo los de control y alimentación, número que se acerca al límite admitido por las tecnologías de encapsulado actuales, que no admiten mucho más que 1.000 terminales por circuito. Una memoria asociativa útil contendría miles de estos nodos, con trayectos de datos más anchos (de mayor cantidad de bits), por lo que el límite del ancho de banda de entrada-salida se alcanza en una etapa muy temprana del diseño de la memoria. Se pueden plantear soluciones de compromiso por medio del multiplexado de los datos sobre el limitado número de conexiones de entrada-salida, pero esta solución reduce la velocidad efectiva, razón primaria para el uso de una memoria asociativa.

Si bien las implementaciones de memorias de direccionamiento por contenido resultan complejas, encuentran aplicaciones prácticas como las mencionadas, referidas a *buffers* de traducción anticipada y en redes de computadoras. En este último caso, una aplicación tiene que ver con controladores de red que reciben paquetes de datos desde diversos procesadores y los distribuyen nuevamente a los procesadores o a otros controladores de red. Cada procesador tiene una dirección única, que la memoria asociativa utiliza como clave para determinar si el procesador de destino de un paquete pertenece a su propia red o si la información debe ser enviada a otra red.

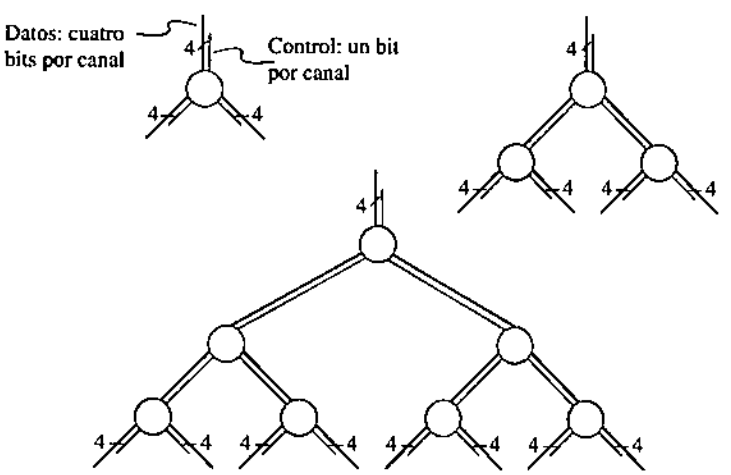


Figura 7.33 • Árboles de direccionamiento para una memoria asociativa.

Ejemplo de diseño de una memoria: memoria RAM de doble puerto

Una memoria **de acceso aleatorio de doble puerto**, o **de lectura dual**, permite la lectura de dos palabras cualesquiera en forma simultánea desde la misma memoria. Como ejemplo, se procederá al diseño de una memoria de doble puerto de 2^{20} palabras de 8 bits. En el planteo se aceptará la lectura simultánea de dos palabras cualesquiera, aun cuando solo se podrá escribir una palabra por vez. El criterio a utilizar es el de construir dos memorias separadas de 2^{20} palabras. Cuando se escribe un dato en la memoria, las líneas de dirección de ambas memorias de acceso aleatorio se establecen en forma idéntica, por lo que se escribe el mismo dato en ambas memorias de lectura única. En la operación de lectura, las líneas de dirección de ambas memorias de lectura única se manejan en forma independiente, por lo que pueden leerse en forma simultánea dos palabras distintas.

La figura 7.34 ilustra un diagrama en bloques de la memoria de acceso aleatorio de lectura dual. Durante la operación de escritura se utilizan las líneas A de direccionamiento para ambas memorias individuales. Las líneas de direccionamiento B presentan *buffers* de tres estados controlados por la señal *WR*. Cuando $WR = 0$, se utilizan las líneas A en las entradas B. En caso contrario, se utilizan las líneas de direccionamiento B. Los números que aparecen adyacentes a las barras diagonales indican la cantidad de líneas individuales representadas por la línea dibujada. Un 8 adyacente a una barra implica 8 líneas individuales, en tanto que un 20 cercano a una barra indica 20 líneas.

Cada *buffer* de tres estados tiene 20 líneas de entrada y 20 líneas de salida. La figura 7.34 utiliza una notación en la que un único esquema representa los 20 *buffers* individuales que comparten la misma línea de control. Sobre la línea *WR* se hace necesario insertar un retardo para compensar el retardo de la línea $\overline{WR}$, de modo que no se produzca una habilitación simultánea no intencional de ambas líneas de direccionamiento A y B. ◦

7.9 Estudio de un caso: la memoria Rambus

En alguna época pasada, la tecnología de la computación de alguna manera se introducía en el mercado desde los laboratorios. A medida que se produjo la explosión del mercado de consumo de equipos de computación, el "empujar" desde lo tecnológico se fue convirtiendo en una exigencia proveniente del mercado, con lo que, cuando surgía la necesidad de desarrollar una tecnología nueva de memorias, las preferencias de los creadores de tecnología resultaban dominadas por la demanda de los consumidores. Los requerimientos de memorias caras, de alto rendimiento, para procesadores de grandes prestaciones fueron reemplazados por memorias de alta densidad y bajo costo para la electrónica de consumo, tal como la de los videojuegos. Los fabricantes de memorias descubrieron que resultaba más redituable satisfacer las necesidades del mercado de consumo de alto

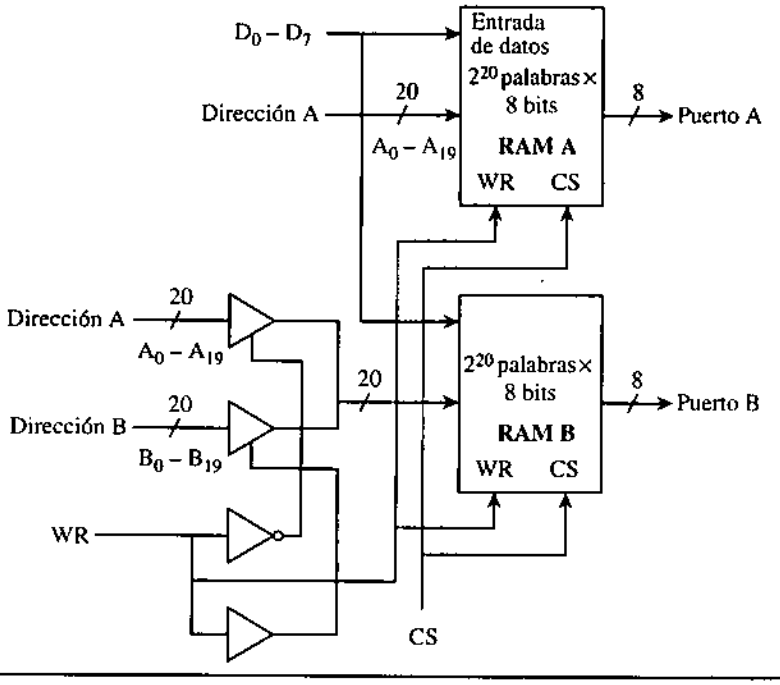


Figura 7.34 • Diagrama en bloques de una memoria de lectura dual.

volumen antes que dedicar costosas plantas de fabricación de circuitos integrados a los mercados comparativamente pequeños de los productos de alta tecnología.

La electrónica de consumo ahora domina el mercado de memorias y aun los procesadores de mayor tecnología hacen uso intenso de las tecnologías electrónicas orientadas a los mercados de consumo, tratando de explotar las mejoras arquitectónicas o innovando en las tecnologías de soporte (tal como las interconexiones de alta velocidad) para compensar las limitaciones de rendimiento de lo que puede llamarse "memorias para videojuegos".

De todos modos, la memoria para videojuegos no constituye el menor nivel y, de hecho, hace uso de mejoras tecnológicas extraordinarias que extraen el mayor rendimiento posible de dispositivos cada vez más densos y de menor costo. Una tecnología de memorias que Intel comenzó a introducir en sus computadoras personales, en 1999, fue desarrollada por Rambus Inc. La memoria dinámica de acceso aleatorio desarrollada por Rambus (RDRAM) rescata un bloque de 8 bytes interno al circuito integrado de memoria con cada acceso, y multiplexa los ocho bytes sobre un canal angosto de 8 o 16 bits, operando a una frecuencia de 800 MHz o superior.

La matriz típica de una memoria DRAM (es decir, la porción de memoria que realiza el almacenamiento en una memoria RAM dinámica común) puede almacenar o recuperar una línea de 8 bytes con una frecuencia de 100 MHz. Este planteo es interno al circuito de memoria: la mayoría de las memorias dinámicas pueden despachar solo un byte por ciclo, pero la tecnología RDRAM puede multiplexar hasta un byte por ciclo usando una

señal de reloj externa de 800 MHz. Esta frecuencia más elevada se inyecta en un controlador de memoria (el "*chipset*" de una máquina Intel) que decodifica los datos sobre un canal de datos de 32 bits a una frecuencia menor, por ejemplo 200 MHz, que lo ingrese en un Pentium (u otro procesador).

Los módulos de memoria RIMM fabricados por Rambus (*Rambus Inline Memory Modules*) tienen un aspecto similar al de los módulos convencionales conocidos como SIMM (*Single Inline Memory Module*) y DIMM (*Double Inline Memory Module*), pero operan en forma diferente. La memoria Rambus utiliza una **tecnología de líneas de transmisión** sobre la placa madre, e implementa un blindaje que reduce los efectos de la **radiofrecuencia** que puede interferir con los datos que circulan a través de las **trazas** de la placa madre. En el diseño de circuitos impresos para la tecnología Rambus, los parámetros críticos son: (1) el espesor dieléctrico de la placa de circuito impreso, (2) la separación de los módulos de memoria y (3) el ancho de las trazas. Debe haber un plano de tierra (un camino eléctrico de retorno) asociado con cada línea de señal, sin interconexiones entre las distintas capas del circuito impreso a lo largo del camino. Todas las señales se ubican en la capa superior del circuito impreso. (El circuito impreso puede tener una cantidad de capas o planos generalmente no mayor que ocho.) El controlador de memoria y los módulos de memoria deben estar todos igualmente espaciados, como, por ejemplo, 1,25 cm desde el controlador de memoria al primer módulo, luego 1,25 cm desde el primer módulo al segundo, etc.

El canal Rambus está constituido por líneas de transmisión. Las trazas terminan siendo bastante más anchas que las trazas convencionales, del orden de los 300 micrones. Si bien 300 micrones representan un ancho relativamente pequeño para una traza de un circuito impreso, si se desean enviar 128 señales a través de un circuito impreso usando un paso (espacio entre centros) de 600 micrones, con trazas de 300 micrones, separados por 300 micrones, se obtiene una huella de 128 x 0,6 mm = 76 mm. Este trazo es relativamente grande en comparación con soluciones de menor velocidad que admiten una densidad de montaje bastante mayor.

En realidad, el canal Rambus solo tiene 13 señales de alta velocidad (la línea de direcciones se serializa en una única línea, hay 8 líneas de datos, 1 línea de paridad, 2 líneas de reloj y 1 línea de comando) y, por consiguiente, la huella aparentemente grande no parece ser un problema de corto plazo. Con una versión del canal Rambus de 16 bit en el horizonte, el problema del ancho de banda parece estar controlado, al menos por algunos años de uso de esta tecnología. Sin embargo, la extensión a palabras de 64 o 128 bits provocará un desafío importante debido a que el conjunto de circuitos integrados deberá permitir el uso de ese mismo tamaño de palabra, tarea formidable para los métodos de encapsulado actuales, que permiten alrededor de 500 terminales en los circuitos.

Si bien las memorias Rambus de este tipo aparecieron y están disponibles desde 1998, los módulos RIMM comenzaron a circular recién en 1999 debido a la necesidad de disponer de un nuevo controlador de memoria para los mismos. El controlador de memoria

es un aspecto importante porque la imagen que percibe el procesador de este tipo de memorias difiere de la percepción de las memorias físicas.

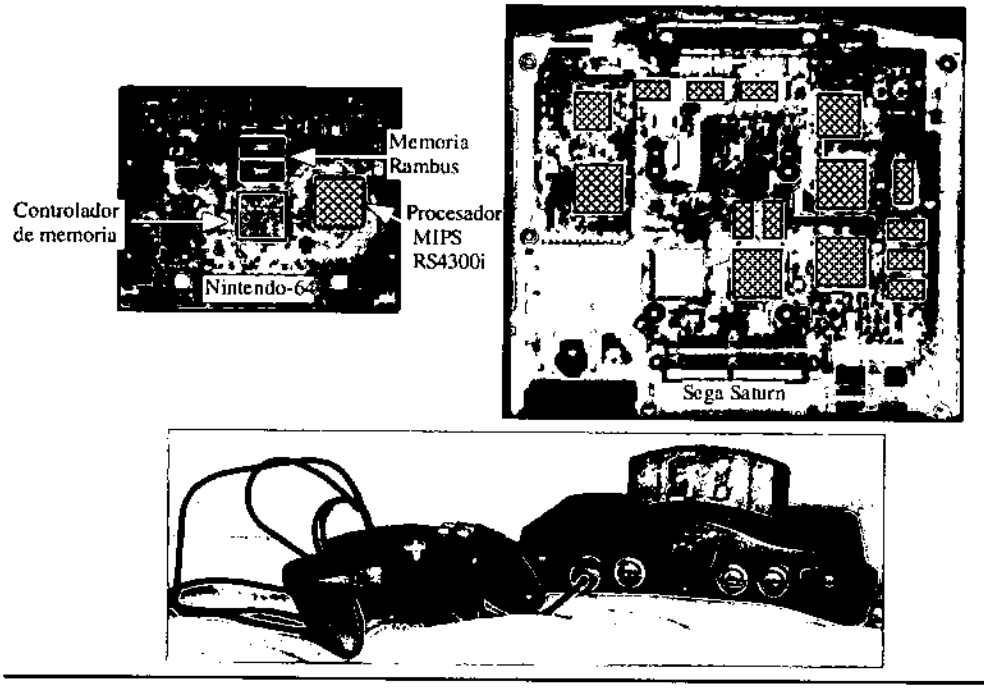


Figura 7.35 • La tecnología Rambus en la placa madre de Nintendo 64 (arriba izquierda) genera ahorros económicos con respecto al diseño convencional Sega Saturn (arriba derecha). Nintendo 64 usa cartuchos externos más costosos (abajo) que los CD-ROM que utiliza Sega Saturn.

La memoria Rambus es más costosa que las memorias DRAM convencionales, pero el costo total del sistema puede reducirse, lo que las hace interesantes en la electrónica de consumo de bajo costo y alto rendimiento, como las consolas de videojuego del tipo de las Nintendo 64. Esta consola (véase la figura 7.35) tiene cuatro circuitos básicos: un procesador MIPS RS4300i de 64 bits, un coprocesador Reality que integra todas las funciones de administración de memoria, gráficas y de sonido, y dos circuitos integrados de memoria Rambus.

La tecnología Rambus provee a la consola Nintendo 64 de un ancho de banda de 562,5 Mbytes/s mediante la utilización de una interfaz de 31 terminales hacia el controlador de memoria. Como elemento comparativo, un sistema que utiliza memorias DRAM sincrónicas de 64 bits requiere una interfaz de 110 terminales con el controlador de memoria. Esta reducción en la cantidad de terminales permite que el controlador de memoria quepa en el mismo elemento de silicio (el circuito integrado en sí) que contiene las funciones gráficas y de sonido, en un circuito integrado de relativamente bajo costo y encapsulado de 160 patas.

El subsistema de memoria Rambus está constituido por dos circuitos integrados de memoria que ocupan alrededor de 10 cm^2 de espacio en el circuito impreso. Un diseño

equivalente en la tecnología SDRAM requerirá cerca de 40 cm^2 de placa de circuito impreso. El ahorro de espacio producido por la solución Rambus le permitió a Nintendo colocar todos sus componentes en una placa de 13 cm por 15 cm, que es alrededor del 25% del tamaño de la placa utilizada por su competidor Sega en sus sistemas Saturn. Además, Nintendo pudo utilizar un circuito impreso de dos capas en lugar del circuito de cuatro capas utilizado por Sega Saturn.

Los ahorros de costos obtenidos por Nintendo al adoptar la solución Rambus, por sobre la utilización de memorias SDRAM de 64 bits, son considerables, pero deben mirarse desde la perspectiva del mercado en su conjunto. La capacidad de implementar su sistema en dos capas le permitió a Nintendo ahorrar 5 dólares por unidad en los costos de fabricación. En el conjunto, la estimación de Nintendo da un ahorro total de alrededor del 20% de la facturación total de materiales con respecto a la solución equivalente con memorias SDRAM.

No obstante, estos ahorros deben evaluarse en función del mercado. Los competidores Sega Saturn y Sony Playstation utilizan CD-ROM para el almacenamiento de los juegos, los que cuestan mucho menos de 2 dólares cuando su producción es masiva. Nintendo 64 utiliza un cartucho de memoria ROM enchufable que cuesta, en producción masiva, alrededor de 10 dólares, y que puede almacenar cerca del 1% de lo que almacena un CD-ROM. Esta selección del medio físico de almacenamiento puede tener un gran impacto sobre la arquitectura general del sistema, por lo que la solución Rambus puede no beneficiar a todos los sistemas en el mismo nivel. Cuando se debe evaluar el impacto de una nueva tecnología sobre un mercado particular, o aun sobre un segmento de un mercado, cualquier detalle puede tener una importancia fundamental sobre el resultado de dicha evaluación.

7.10 Estudio de un caso: el sistema de memoria Pentium de Intel

El procesador Pentium de Intel tiene una configuración de memoria que es habitual en los procesadores modernos. La figura 7.36 muestra un diagrama simplificado de los elementos de memoria y de los trayectos de datos. Existen dos memorias *cache* de nivel 1 incorporadas en el circuito integrado, una memoria *cache* para instrucciones, llamada I *cache*, y una memoria *cache* de datos, D-*cache*. Cada *cache* tiene un tamaño de línea de 256 bits (32 bytes), con un trayecto de datos hacia la CPU de igual tamaño. Las dos memorias *cache* de nivel 1 son asociativas por conjuntos, de dos vías, y cada vía tiene un único bit LRU. La memoria *cache* de datos puede hacerse funcionar con escritura inmediata o diferida para cada línea en forma independiente. Esta memoria trabaja en forma de escritura sin asignación (*write-no-allocate*). Las fallas de escritura no generan como resultado el llenado de la memoria *cache*. Cada *cache* está equipada, además, con un *buffer* de traducción anticipada (TLB) que convierte las direcciones virtuales en direcciones físicas. El *buffer* de traducción anticipada de la memoria *cache* de datos es de cuatro vías, asociativa

por conjuntos, con 64 entradas, con dos puertos, por lo que permite la traducción simultánea de dos referencias de datos. El TLB de la memoria *cache* de instrucciones es de cuatro vías, asociativa por conjuntos, con 32 entradas.

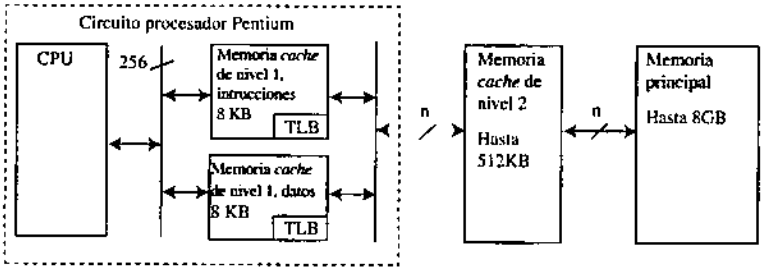


Figura 7.36 • El sistema de memoria Pentium de Intel.

Si existe memoria *cache* de nivel 2, la misma es asociativa por conjuntos, de dos vías, pudiendo ser de 256 KB o de 512 KB. El bus de datos, que se indica en la figura como *n*, puede tener un tamaño de 32, 64 o 128 bits.

El protocolo MESI

La memoria *cache* de datos del procesador Pentium y la memoria de nivel 2, si la hubiera, utilizan el protocolo de coherencia MESI para administrar accesos multiprocesadores a memoria. Cada línea de la memoria *cache* de datos tiene asociados dos bits que se utilizan para almacenar el estado MESI. Cada una de las líneas de la memoria *cache* podrá estar en uno de los siguientes estados:

- M - Modificado: el contenido de la línea de *cache* ha sido modificado y difiere del contenido de la memoria principal.
- E - Exclusivo: el contenido de la línea de *cache* no ha sido modificado y coincide con el contenido de la memoria principal.
- S - Compartido: la línea está, o puede estar, compartida con alguna otra línea de memoria *cache* que pertenece a otro procesador.
- I - Inválido: la línea no está en la memoria *cache*. La lectura de líneas en estado I dará por resultado una falla de *cache*.

La tabla 7.2[1] muestra la relación entre el estado MESI, la línea de memoria *cache* y la línea equivalente de memoria principal. El protocolo MESI se está convirtiendo en una forma normalizada de tratamiento de la coherencia de las memorias *cache* en sistemas multiprocesadores.

1. Tomada de *Pentium Processor User's Manual*, vol. 3, *Architecture and Programming Manual*, © Intel Corporation, 1993.

Estado de la línea de memoria *cache*	M Modificado	E Exclusivo	S Compartido	I Inválido
¿La línea de memoria *cache* es válida?	Sí	Sí	Sí	No
La copia en memoria principal es	Obsoleta	Válida	Válida	—
¿Existen copias en otras memorias *cache*?	No	No	Puede ser	Puede ser
Una escritura hacia esa línea	No sale al bus	No sale al bus	Sale directamente al bus y actualiza la memoria *cache*	Sale directamente al bus

Tabla 7.2 • Estados de líneas de memoria *cache* según protocolo MESI.

El procesador Pentium soporta, además, hasta seis segmentos de memoria principal (puede haber varios miles de segmentos, pero no se puede hacer referencia a más de seis de ellos a través de los registros de segmentos). De acuerdo con el análisis realizado a lo largo del capítulo, cada segmento es un espacio de direcciones diferente. Cada segmento tiene una base, es decir, una dirección de comienzo dentro del espacio de direcciones físicas de 32 bits, y un límite, el tamaño máximo del segmento. El límite puede tener un tamaño que bien puede ser de 2^{16} bytes o bien de $2^{16} \times 2^{12}$ bytes. Esto es, la granularidad del límite puede tener un tamaño de un byte o de 2^{12} bytes.

Las técnicas de paginación y segmentación se pueden utilizar en procesadores Pentium, en cualquier combinación:

- Memoria sin segmentación ni paginación: el espacio de direcciones virtuales coincide con el espacio de direcciones físicas. No se requieren tablas de páginas ni hardware para asignaciones. Es una configuración adecuada para aplicaciones de alto rendimiento que no tienen una gran complejidad (por ejemplo, no necesitan soportar tablas crecientes).
- Memoria paginada, sin segmentación: igual que en el caso anterior, excepto porque tiene un espacio lineal de direcciones mayor como resultado del uso de almacenamiento en disco. Se requiere una tabla de páginas para la traducción entre direcciones virtuales y físicas. Se requiere un *buffer* de traducción anticipada en el procesador, el que debe trabajar en conjunción con la memoria *cache* de nivel 1, con el objeto de reducir la cantidad de accesos a las tablas de páginas.
- Memoria segmentada, sin paginación: sirve para aplicaciones de alta complejidad que necesitan soportar estructuras de datos crecientes. También es rápida: la cantidad de segmentos es menor que la cantidad de páginas en una memoria virtual y, además, toda la electrónica de asignación para la segmentación cabe dentro del circuito integrado del procesador. No hay necesidad de accesos a disco, como habría en el caso de la paginación, por lo que los tiempos de acceso son más predecibles que cuando se utiliza la paginación.

- Memoria segmentada y paginada: se requiere el uso de una tabla de páginas, de registros de asignación de segmentos y de un *buffer* de traducción anticipada que funcionen en forma armónica para poder manejar múltiples espacios de direcciones.

La técnica de segmentación es muy poderosa en el procesador Pentium Intel , pero también es bastante compleja. Se ha hecho aquí solo una revisión de los conceptos básicos, por lo que el lector interesado puede referirse al texto de Intel para una descripción más completa.

Resumen

La memoria de un sistema se encuentra organizada en forma jerárquica, donde la memoria más densa ofrece el peor de los rendimientos, en tanto que el mejor rendimiento se obtiene con la memoria de menor densidad. Con el objeto de salvar la brecha entre ambas, se aprovecha el principio de localidad en el uso de memorias *cache* y de memorias virtuales.

Una memoria *cache* mantiene los bloques usados con mayor frecuencia en una memoria pequeña y rápida que se encuentra localizada en la CPU. La memoria virtual paginada aumenta el tamaño de la memoria principal a través del almacenamiento en disco. La memoria física sirve como ventana en la memoria paginada virtual, la que se mantiene en forma íntegra en un disco magnético.

Las memorias *cache* y virtual paginada se utilizan habitualmente en la misma computadora, pero por razones diferentes. La memoria *cache* mejora el tiempo medio de acceso a la memoria principal, en tanto que la memoria virtual paginada extiende el tamaño de la memoria principal.

En un ejemplo de arquitectura de memoria, el procesador Pentium de Intel posee una memoria *cache* de nivel 1 dividida y un registro de traducción anticipada que reside en el procesador Pentium, y tiene una memoria *cache* de nivel 2 unificada que reside en el mismo encapsulado que el procesador, pero en una pastilla de silicio diferente. Cuando se implementa la paginación, la tabla de páginas se ubica en memoria principal y se utiliza el *buffer* de traducción anticipada para disminuir la cantidad de veces que se hace referencia a la tabla de páginas. Las memorias *cache* de nivel 1 y 2 reducen la cantidad de veces que se accede a la memoria principal en busca de datos. El procesador Pentium también acepta el mecanismo de segmentación, el que tiene su propio conjunto de registros de asignación y electrónica de control residentes en el procesador Pentium.

Para lectura posterior

W. Stallings y M. Mano brindan una explicación muy clara de las memorias de acceso aleatorio. Una cantidad de manuales de memoria (Micron y Texas Instruments) ofrecen ejemplos prácticos de organizaciones de memoria. C. C. Foster es el referente principal en el tema de memorias asociativas. C. Mead y L. Conway describen la estructura del árbol H en el contexto del diseño con técnicas

VLSI. M. A. Franklin y otros analiza las cuestiones que surgen en los circuitos de partición, las que aparecen en la división de un árbol H para una memoria asociativa. A. Sedra y K. Smith analizan la implementación de distintos tipos de memorias de acceso aleatorio estáticas y dinámicas.

V. C. Hamacher y otros presenta un tratamiento clásico de las memorias *cache*. A. Tanenbaum ofrece una explicación clara sobre el tema de memoria virtual. J. L. Hennessy y D. A. Patterson, y S. A. Przybylski cubren los temas asociados con el rendimiento de las memorias *cache*. La técnica de segmentación aplicada al procesador Pentium Intel se encuentra cubierta en el texto de Intel. Kingston Technology ofrece un curso amplio sobre tecnologías de memorias en la dirección http://www.kingston.com/king/mg0.htm.

Foster, C. C., *Content Addressable Parallel Processors*, Van Nostrand Reinhold Company, 1976.

Franklin, M. A., D. F. Wann y W. J. Thomas, "Pin Limitations and Partitioning of VLSI Interconnection Networks", en: *IEEE Transactions on Computers*, C-31, 1109, noviembre de 1982.

Hamacher, V. C., Z. G. Vranesic y S. G. Zaky, *Computer Organization*, 3ª ed., McGraw-Hill,1990.

Hennessy, J. L. y D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2ª ed., Morgan Kaufmann Publishers, 1995.

Intel Corporation, *Pentium Processor User's Manual*, vol. 3: *Architecture and Programming Manual*, 1993.

Knuth, D., *The Art of Computer Programming: Fundamental Algorithms*, vol. 1, 2ª ed., Addison Wesley, 1974.

Mano, M., *Digital Design*, 2ª ed., Prentice Hall, 1991. (Traducción al español disponible: *Diseño digital*, Prentice Hall.

Mead, C. y L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

Micron, *DRAM Data Book*, Micron Technologies, Inc., 2805 East Columbia Road, Boise, Idaho, 1992.

Przybylski, S. A., *Cache and Memory Hierarchy Design*, Morgan Kaufmann Publishers, 1990.

Sedra, A. y K. Smith, *Microelectronic Circuits*, 4ª ed., Oxford University Press, 1997.

Stallings, W., *Computer Organization and Architecture*, 3ª ed., Macmillan, 1993. (Traducción disponible al español: *Organización y arquitectura de computadores*, 5ª ed., Prentice Hall, 2000.)

Tanenbaum, A., *Structured Computer Organization*, 3ª ed., Prentice Hall, 1990. (Traducción al español disponible: *Organización de computadoras*, 4ª ed., Prentice Hall, 2000.)

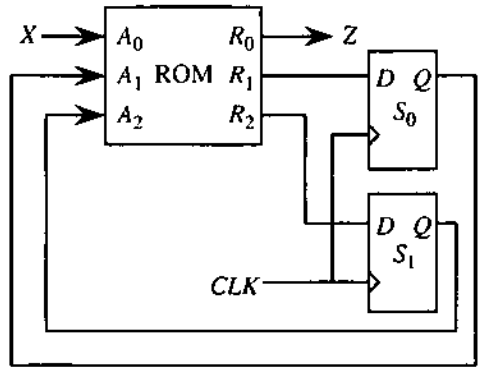
Texas Instruments, *MOS Memory: Commercial and Military Specifications Data Book*, Texas Instruments, Literature Response Center, P. O. Box 172228, Denver, Colorado, 1991.

Problemas

7.1 Una tabla de busquedas en memoria ROM y dos *flip flops* D implementan una máquina de estados como la que se muestra en el diagrama. Construir una tabla de estados que describa el comportamiento de la máquina.

Contenido de la memoria de lectura

Ubicación			Contenido		
A_0	A_1	A_2	R_0	R_1	R_2
0	0	0	0	0	1
0	0	1	1	1	0
0	1	0	1	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	0	0	1
1	1	1	0	0	0



7.2 Para la tabla de busquedas de la figura 7.11, completar cuatro direcciones de memoria que permitan realizar las operaciones de suma, resta, producto y cociente sobre $A = 16$ y $B = 4$. Indicar la dirección y el valor correspondientes a cada caso.

7.3 Diseñar una memoria RAM de 32 palabras de ocho bits, usando circuitos de memoria de 8 x 8.

7.4 Diseñar una memoria RAM de 16 palabras de cuatro bits, usando circuitos de 4 x 4 y un solo decodificador externo.

7.5 Dada una cierta cantidad de circuitos integrados de memoria RAM de n palabras de p bits,

a. Indicar cómo puede construirse con estos circuitos integrados una memoria RAM de n palabras por $4p$ bits. Utilizar cualquier otro elemento lógico de los que se han empleado en el texto según se considere necesario.

b. Indicar cómo puede construirse con estos circuitos integrados una memoria RAM de $4n$ palabras por p bits.

7.6 Dibujar el circuito de un árbol decodificador de 4 a 16, que utiliza como *fan in* y *fan out* máximos el valor 2.

7.7 Una memoria *cache* de asignación directa consiste en 128 líneas. La memoria principal contiene 16 Kbloques de 16 palabras cada uno. El tiempo de acceso de la memoria *cache* es de 10 ns y el tiempo requerido para completar una línea de la memoria *cache* es de 200 ns. No se utiliza carga inmediata. Cuando se intenta acceder a una palabra que no se encuentra en la memoria *cache*, se carga un bloque íntegro en la memoria *cache*, tras lo cual se accede a la palabra desde la memoria *cache*. En su estado inicial, la memoria *cache* está vacía. No olvidar que en las referencias en memoria, 1 K = 1024.

a. Indicar el formato de la dirección de memoria.

b. Calcular la tasa de aciertos de un programa que se ejecuta en forma repetitiva diez veces desde la dirección 15 a la dirección 200. Nótese que aunque la memoria se accede dos veces en caso de una falla (una en el momento de la falla y otra para satisfacer el requerimiento), no se produce acierto en este caso. El programa en ejecución observa una sola referencia a memoria.
c. Calcular el tiempo de acceso efectivo del programa mencionado.

7.8 Una memoria *cache* totalmente asociativa tiene 16 bloques, con ocho palabras por bloque. El tamaño de la memoria principal es de 2^{16} palabras y la memoria *cache* inicialmente está vacía. El tiempo de acceso de la memoria *cache* es de 40 ns, y el tiempo requerido para transferir ocho palabras entre memoria principal y memoria *cache* es de 1 μs.
a. Calcular los tamaños de los campos de etiqueta y palabra.
b. Calcular la tasa de aciertos de un programa que se ejecuta desde la dirección 20 a la dirección 45, tras lo cual se ejecuta cuatro veces desde 28 hasta 45 antes de detenerse. Suponer que, en caso de una falla, la línea completa de memoria *cache* se completa en 1 μs y que la primera palabra no puede ser vista desde el procesador hasta que se complete íntegramente la línea. Esto significa suponer que no se realiza carga inmediata. La memoria *cache* está inicialmente vacía.
c. Calcular el tiempo efectivo de acceso del programa descripto en b.

7.9 Calcular la cantidad total de bits de almacenamiento que se requieren para la memoria *cache* asociativa de la figura 7.13 y para la memoria *cache* de asignación directa de la figura 7.14. Incluir en el cálculo los bits de validez, etiquetas y suciedad. Suponer que el tamaño de la palabra es de ocho bits.

7.10 a. ¿A qué distancia deben espaciarse las referencias a memoria para que provoquen una falla en cada acceso a memoria *cache* utilizando los parámetros de asignación directa de la figura 7.14?
b. Utilizando la respuesta de la parte (a), calcular la tasa de aciertos y el tiempo efectivo de acceso de un programa considerando los tiempos de carga en caso de acierto y de falla del problema 7.8 ($T_{acierto}$ = 10 ns, T_{falla} = 1.000 ns). Suponer que en este caso se utiliza carga inmediata.

7.11 Una computadora tiene 16 páginas de espacio virtual de direcciones pero solo tiene cuatro marcos de página física. En el estado inicial, la memoria física está vacía. Un programa hace referencia a las páginas virtuales en el orden 0 2 4 5 2 4 3 11 2 10.
a. ¿Cuáles son las referencias que provocan falla de página con el criterio LRU de reemplazo de páginas?
b. ¿Cuáles son las referencias que provocan falla de página con el criterio FIFO de reemplazo de páginas?

7.12 En algunas computadoras, la tabla de páginas se almacena en memoria. ¿Qué ocurriría si la tabla de páginas se transfiere hacia el disco? Dado que la tabla de páginas se utiliza en cada re-

ferencia de memoria, ¿hay algún criterio de reemplazo de páginas que garantice que la tabla de páginas no será eliminada de la memoria? Suponer que la tabla de páginas es bastante chica como para caber en una sola página (aunque habitualmente no lo sea).

7.13 Un sistema de memoria virtual tiene un tamaño de página de 1024 palabras, ocho páginas virtuales, cuatro marcos de página física, y utiliza el criterio LRU de reemplazo de páginas. La tabla de páginas tiene el formato que se ilustra:

Página #	Bit de presencia	Dirección de disco	Campo del marco de páginas
0	0	01001011100	xx
1	0	11101110010	xx
2	1	10110010111	00
3	0	00001001111	xx
4	1	01011100101	01
5	0	10100111001	xx
6	1	00110101100	11
7	0	01010001011	xx

¿Cuál es la dirección de memoria principal correspondiente a la dirección virtual 4096?
¿Cuál es la dirección de memoria principal correspondiente a la dirección virtual 1024?
Se produce una falla en la página 0. ¿Cuál de los marcos de página se utiliza para la página virtual 0?

7.14 Cuando se ejecuta un programa específico con N accesos a memoria, una computadora que tiene memoria *cache* y memoria virtual paginada genera un total de M fallas de la memoria *cache* y F fallas de página. T_1 es el tiempo en caso de un acierto de memoria *cache*, T_2 es el tiempo en caso de un acierto de memoria principal y T_3 es el tiempo requerido para cargar una página desde el disco a la memoria principal.

a. ¿Cuál es la tasa de aciertos de memoria *cache*?
b. ¿Cuál es la tasa de aciertos de memoria principal? Esto es, ¿qué porcentaje de los accesos a memoria principal no generan falla de página?
c. ¿Cuál es el tiempo efectivo de acceso del sistema?

7.15 Una computadora contiene tanto memoria *cache* como memoria virtual paginada. La memoria *cache* puede contener tanto direcciones físicas como virtuales, pero no ambas. ¿Cuáles son los elementos a tener en cuenta para decidir si se colocan en la memoria *cache* direcciones virtuales o físicas? ¿Cómo pueden resolverse estos problemas por medio del uso de un elemento único que administre todas las funciones de asignación de memoria?

7.16 ¿Qué cantidad de memoria se requiere para la tabla de páginas de una memoria virtual de 2^{32} bytes, con 2^{12} bytes por página y 8 bytes por entrada a la tabla de páginas?

7.17 Calcular la cantidad de entradas de compuertas para el (los) decodificador(es) de una memoria RAM de 64 x 1 bit, tanto con las técnicas 2D y 2-1/2D. Suponer que se admiten *fan out* y *fan in* ilimitados. Para ambos casos, utilizar decodificadores convencionales de dos niveles. Para el caso 2-1/2D, tratar el decodificador de columnas como un multiplexor convencional. Esto implica ignorar su comportamiento como demultiplexor durante las operaciones de escritura.

7.18 ¿Cuántos niveles de decodificación se requieren para una memoria 2D de 2^{20} palabras si se admite un número máximo de entradas y salidas de cuatro en el árbol de decodificación?

7.19 Un cartucho de videojuego requiere almacenar 2^{20} palabras en una memoria ROM.
a. ¿Si se utiliza una organización 2D, cuántas hojas habrá en el último nivel del árbol decodificador?
b. ¿Cuántas hojas habrá en el último nivel del árbol decodificador si la organización es 2-1/2D?

7.20 La tabla muestra el contenido de una memoria asociativa. Si se utiliza una clave de 00A00020 en los campos 1 y 3, ¿qué conjunto de palabras responderá? Los campos 1 y 3 de la clave deben coincidir con los campos correspondientes de una palabra de la memoria para que esa palabra responda. Los campos restantes se ignoran durante el proceso de verificación de coincidencia pero se incluyen en las palabras recuperadas.

Campo → 4	3	2	1	0
F 1	A	0 0	0 2	B
0 4	2	9 D	1 F	0
3 2	A	1 1	0 3	E
D F	A	0 5	0 2	D
0 0	5	3 7	F 2	4

7.21 Cuando el *buffer* de traducción anticipada de la figura 7.27 produce una falla, accede a la tabla de páginas para resolver la referencia. ¿Cuántas entradas hay en esa tabla de páginas?

Capítulo 8

Entrada y salida

En los capítulos anteriores se ha analizado la forma en que la CPU interactúa con datos que se encuentran almacenados dentro de la misma o en la memoria principal, lo que puede extenderse a una unidad de disco magnético a través del concepto de memoria virtual. Si bien los tiempos de acceso de los distintos niveles de la jerarquía de memoria varían en forma apreciable, en la mayor parte de los casos la CPU ve la misma velocidad de respuesta en los diferentes accesos. La situación es muy diferente cuando se accede a dispositivos de entrada-salida.

- La velocidad de las transferencias de entrada-salida puede variar desde una lentitud extrema, como ocurre cuando se ingresan datos desde un teclado, hasta una velocidad tan alta que la CPU puede no ser capaz de seguirle el paso, como ocurre con el flujo de datos que ingresa desde una unidad de disco muy rápida, o con la salida de gráficos en tiempo real hacia un monitor de video.
- Las actividades de entrada-salida son **asincrónicas**. Esto significa que no están sincronizadas con el reloj de la CPU como sí lo están las transferencias de memoria. Con el objeto de coordinar la disponibilidad del dispositivo de entrada-salida, para permitir la lectura o escritura de la información destinada al mismo, puede presentarse la necesidad de incorporar señales adicionales, en un bus de entrada-salida específico.*
- La calidad de la información puede ser sospechosa. Por ejemplo, el ruido de línea durante las transferencias de datos a través de las redes conmutadas de telefonía pública o los errores producidos por defectos en la superficie de un disco plantean la necesidad de introducir estrategias de detección y corrección de errores para asegurar la integridad de los datos.
- Muchos dispositivos de entrada-salida son mecánicos y, en general, tienen más probabilidad de fallar que la CPU y la memoria principal. Puede producirse la interrup-

* *N. de T.*: El texto original hace referencia al intercambio de señales conocido como "*handshaking signals*", forma habitual de denominar al protocolo de señales de control que se establece entre la CPU y la interfaz de entrada-salida para la transferencia de datos.

ción de una transferencia de información debido a una falla mecánica o a condiciones especiales, como sucede cuando una impresora se queda sin papel.

- Por consiguiente, se deben tener en cuenta todas estas situaciones durante el desarrollo de los módulos de software destinados al manejo de las transferencias de entrada-salida, conocidos como **controladores de dispositivos** (*device drivers*).

Este capítulo analiza la naturaleza de las comunicaciones que utilizan buses. Comienza con un análisis de los fundamentos de las arquitecturas de bus único y continúa con las de bus múltiple. Se completa el capítulo con una mirada a los dispositivos de entrada-salida más comúnmente conectados a estos buses.

Las secciones siguientes analizan el tema de la transferencia de datos desde los puntos de vista de las comunicaciones en el ámbito de la CPU y de la placa madre, tras lo cual se deriva hacia el concepto de la red de área local.

8.1 Arquitecturas de un único bus

Un sistema de computación puede contener muchos componentes que deban comunicarse con los demás. En el escenario del peor caso, los N componentes de un sistema necesitan comunicarse simultáneamente con algún otro elemento, por lo tanto requieren $N^2/2$ vínculos para los N componentes. La cantidad de vínculos se vuelve prohibitiva aún para valores pequeños de N, pero, afortunadamente, al igual que en el caso de las telecomunicaciones de larga distancia, no todos los dispositivos requieren comunicarse en el mismo momento.

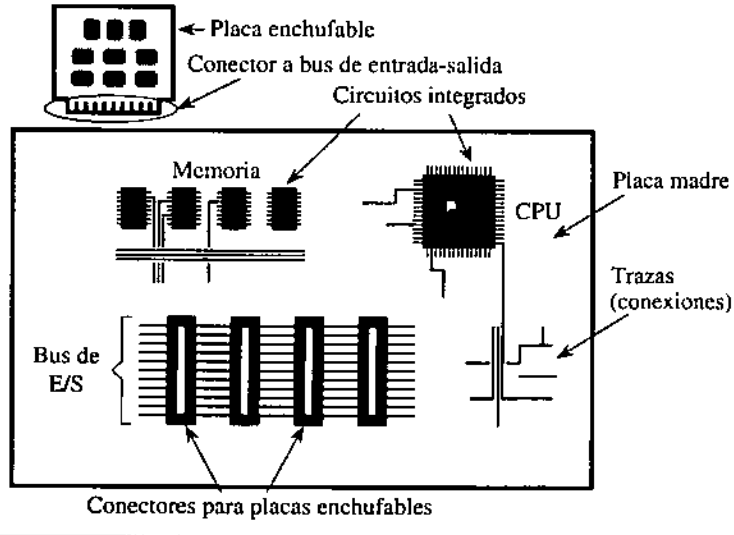


Figura 8.1 • Una placa madre de una computadora personal (vista simplificada desde arriba).

Un **bus** es un camino común que conecta una cantidad de dispositivos. Un ejemplo de bus se puede encontrar en la **placa madre** (*motherboard*, la placa principal de circuito impreso que contiene la unidad de proceso) de una computadora personal, según se ilustra en forma simplificada en la figura 8.1. (La imagen de una placa real puede verse en la figura 1.6). Una placa madre típica incluye **circuitos integrados** tales como el que contiene al procesador y los circuitos de memoria principal, **trazas** (conexiones) que interconectan los circuitos integrados, y una cantidad de buses para satisfacer las necesidades de comunicación de circuitos integrados o dispositivos que requieran comunicarse unos con otros. En la figura 8.1 se utiliza un bus de entrada-salida para conectar un conjunto de placas que se insertan en los conectores, en forma perpendicular a la placa madre en la configuración del ejemplo.

8.1.1 Estructura de bus, protocolos y control

La estructura de un bus no solo contiene elementos físicos, como cables y conectores, sino que incluye además un **protocolo**. Los cables pueden estar divididos en grupos separados de líneas de dirección, control, datos y alimentación, como se muestra en la figura 8.2. Un bus unificado puede tener unas pocas líneas de alimentación, las que en el caso de la figura 8.2 incluyen líneas de tierra (GND) a 0V, y líneas de tensiones positivas y negativas a +5 V y -15 V, respectivamente.*

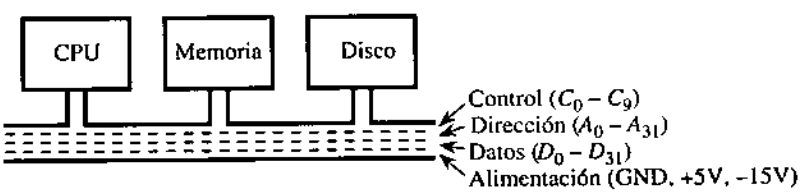


Figura 8.2 • Ilustración simplificada de un bus.

Los dispositivos comparten un conjunto común de cables, a través del cual solo un dispositivo puede enviar información en un momento dado. Todos los dispositivos escuchan a la vez, pero normalmente solo uno de ellos recibe. Nada más que uno de los dispositivos puede ser **maestro del bus** (*bus master*), en tanto que los dispositivos restantes se consideran **esclavos**. El maestro tiene el control del bus y puede ser tanto transmisor como receptor.

El uso de un bus elimina la necesidad de conectar cada dispositivo con los demás, lo que evita la complejidad de interconexión que dominaría inmediatamente el costo de tal sistema. Entre las desventajas del uso de un bus se cuentan la lentitud introducida por la configuración maestro-esclavo, el tiempo que se requiere para implementar un protocolo

* *N. de T.*: En las placas utilizadas actualmente se encuentra una gran variedad de tensiones que dependen del procesador empleado. Entre ellas, +5V, -5V, +12V, -12V y distintas tensiones en el orden de 2,7V a 3,3V.

y la dificultad de escalar el diseño hacia tamaños mayores debido a restricciones de temporizaciones y corrientes.

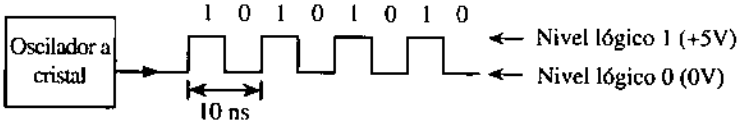


Figura 8.3 • Un reloj para un bus de 100 MHz.

Los buses de un sistema pueden dividirse en dos tipos: **sincrónicos** o **asincrónicos**. Para el caso de un bus sincrónico, uno de los dispositivos conectados al mismo contiene un oscilador (una señal de reloj), que genera una salida de unos y ceros a intervalos regulares, como se muestra en la figura 8.3. La figura ilustra un tren de pulsos que se repiten a intervalos de 10 ns, lo que corresponde a una frecuencia de reloj de 100 MHz. En el caso ideal, la señal de sincronismo debería ser una onda perfectamente cuadrada (con cambios instantáneos entre uno y otro nivel), como se muestra en el diagrama. En la práctica, los flancos de subida y bajada de la señal se pueden asociar con una forma trapezoidal redondeada.

8.1.2 Frecuencias de reloj en el bus

Cuando el bus de un sistema es sincrónico, como el que se describe en las secciones siguientes, se utiliza una señal de reloj para sincronizar las operaciones sobre el mismo. Esta señal de reloj se deriva generalmente desde la señal maestra del reloj del sistema, aun cuando puede ser menor que esa señal maestra, en especial en sistemas con un procesador de alta velocidad. Por ejemplo, una computadora modelo Macintosh G3 tiene una frecuencia de reloj de 333 MHz para el sistema, pero una frecuencia de solo 66 MHz para el bus, o sea, 5 veces menor que la anterior. Esto tiene relación con los tiempos de acceso de memoria, que son mucho mayores que los que se requerirían en función de la frecuencia de funcionamiento interno de la CPU. Una memoria *cache* típica tiene un tiempo de acceso cercano a los 20 ns, el que no puede compararse con el período de 3 ns de la señal de reloj descripta anteriormente.

No solo es necesario que el reloj del bus funcione a menor velocidad que el procesador, sino que normalmente se requieren varios ciclos del reloj para concretar una transacción sobre el bus. El conjunto de ciclos requeridos va de dos a cinco períodos completos de la señal de reloj, y suele ser denominado **ciclo de bus**.

8.1.3 El bus sincrónico

Para ejemplificar cómo se producen las comunicaciones en un bus sincrónico, considérese el diagrama de tiempos de la figura 8.4, correspondiente a la lectura sincrónica de una palabra de memoria, realizada por la CPU. En algún punto inicial del intervalo de tiem-

po T_1, durante el semiciclo positivo de la señal de reloj, la CPU coloca sobre las líneas de dirección del bus la dirección de la posición que quiere leer. En un momento posterior, durante T_1, luego de haberse estabilizado las líneas de direcciones, el procesador activa las líneas $\overline{MREQ}$ y $\overline{RD}$. La línea $\overline{MREQ}$ le indica a la memoria que se encuentra seleccionada para realizar una transferencia, en tanto que la línea $\overline{RD}$ informa que la transacción a realizar es una lectura. Las barras sobre las identificaciones de las señales $\overline{MREQ}$ y $\overline{RD}$ indican que se activan cuando el nivel de las mismas es cero.

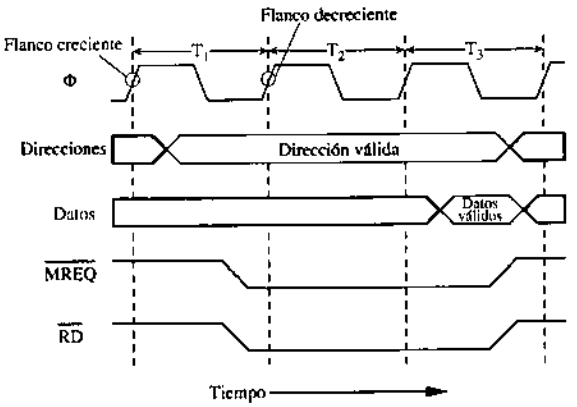


Figura 8.4 • Diagrama de tiempos para una lectura sincrónica de memoria (adaptado de Tanenbaum).

Generalmente, el tiempo de lectura de la memoria es mayor que la velocidad del bus, por lo que la lectura consume completamente el período T_2, así como parte del T_3. La unidad de proceso supone un tiempo fijo de lectura de tres ciclos de reloj, por lo que rescata el dato leído desde el bus durante el tercer ciclo. La CPU libera el bus desactivando en T_3 las señales $\overline{MREQ}$ y $\overline{RD}$. Las zonas sombreadas de los diagramas de datos y direcciones, en el diagrama de tiempos, indican que esas señales, durante esos lapsos, son inválidas, o bien irrelevantes. Las áreas no sombreadas del diagrama, como la que corresponde a las líneas de datos durante T_3, representan señales válidas. En los diagramas se utilizan estas áreas sombreadas y no sombreadas con líneas cruzadas en los extremos para indicar que los niveles de las líneas individuales del bus pueden ser diferentes.

8.1.4 El bus asincrónico

Si se reemplaza la memoria colocada sobre un bus sincrónico con una memoria más rápida, el tiempo de acceso de la memoria no se modificará en tanto no se modifique la frecuencia del reloj del bus. Si se incrementa la frecuencia de la señal de reloj sobre el bus para mejorar las comunicaciones con la memoria, puede ocurrir que dejen de funcionar correctamente algunos dispositivos lentos que utilizan la misma señal de reloj del bus.

Un bus asincrónico resuelve el problema, pero es más complejo debido a que no hay señal de sincronismo. El dispositivo maestro que controla al bus coloca sobre él todo lo que necesita para una transacción (direcciones, datos, control), tras lo cual activa una señal $\overline{MSYN}$ (*master synchronization*). El esclavo realiza su parte del trabajo y, cuando lo termina, activa su propia señal $\overline{SSYN}$ (*slave synchronization*). En ese momento, el maestro inactiva $\overline{MSYN}$, lo que habilita al esclavo a inhibir $\overline{SSYN}$. De esta forma, una combinación maestro-esclavo rápida responde mucho más velozmente que una combinación maestro-esclavo lenta.

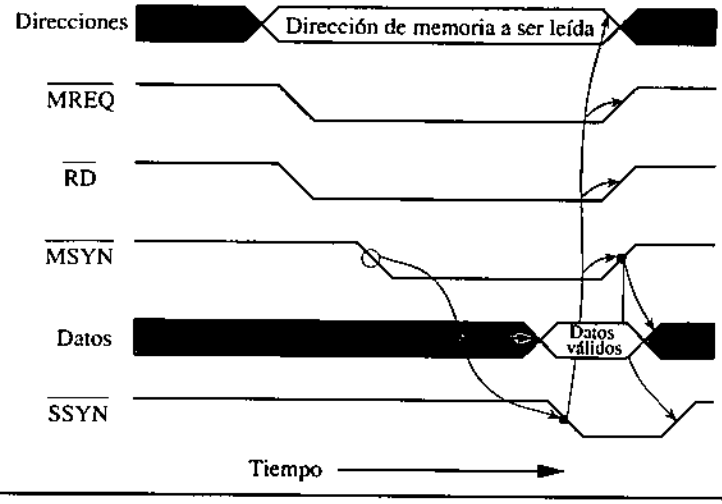


Figura 8.5 • Diagrama de tiempos en la lectura asincrónica de memoria (adaptado de Tanenbaum).

Para analizar la forma en que se establece una comunicación sobre un bus asincrónico, considérese el diagrama de tiempos de la figura 8.5. Con el objeto de leer una palabra desde memoria, el procesador coloca una dirección de memoria sobre el bus, tras lo cual activa las señales $\overline{MREQ}$ y $\overline{RD}$. Después del establecimiento de estas líneas, la CPU activa $\overline{MSYN}$. Este último evento dispara la operación de lectura en la memoria, lo que da por resultado la eventual activación, por parte de la memoria, de la línea $\overline{SSYN}$. Esto se indica en la figura 8.5 a través de las **flechas de causa y efecto** que vinculan a $\overline{MSYN}$ y $\overline{SSYN}$. Este método de sincronización se conoce como proceso de sincronización completa o diálogo completo (*full handshake*). En esta implementación particular de un protocolo asincrónico dialogado, la transferencia se inicia cuando se valida $\overline{MSYN}$, continúa con la validación de $\overline{SSYN}$, luego ocurre la inhibición de $\overline{MSYN}$ por parte de la CPU y, finalmente, la inhibición de $\overline{SSYN}$ por parte de la memoria. Nótese la falta total de una señal de reloj.

En el caso de que exista un problema, un bus asincrónico puede ser más complejo de depurar en su funcionamiento que un bus sincrónico, así como pueden resultar más complejos los diseños de interfaces para un bus asincrónico. Por esta razón, los buses sincrónicos son de uso más común, especialmente en computadoras personales.

8.1.5 Arbitraje del bus. Maestros y esclavos

Supóngase ahora que en un determinado instante hay más de un dispositivo que requiere ser maestro del bus. ¿Cuál es la forma de decidir quién debe ser el maestro? El problema del **arbitraje del bus** ofrece dos esquemas básicos: **centralizado** o **descentralizado** (distribuido). La figura 8.6 muestra cuatro posibles organizaciones de estos dos esquemas. En la figura 8.6a se utiliza un esquema de arbitraje centralizado. Los dispositivos 0 a *n* están todos conectados al mismo bus (que no se muestra) y comparten también una **línea de pedido de uso del bus** (*bus request*) que llega al **árbitro**. Cuando un dispositivo requiere tomar control del bus, activa la línea de pedido. Cuando el pedido llega al árbitro, este determina si puede **conceder** (*bus grant*) el uso del bus (puede ocurrir que el actual maestro del bus no admita que se lo interrumpa). Si se puede otorgar el uso del bus, el árbitro activa la línea de concesión. Esta línea está **encadenada** (*daisy chained*) desde un dispositivo al siguiente. El primer dispositivo, de los que necesitan ser maestro del bus, que reciba la señal de concesión activada tomará control del bus e impedirá que la señal de concesión se propague hacia los dispositivos más lejanos (los de numeración más elevada). Si el dispositivo que recibe la señal no necesita usar el bus, simplemente permite el paso de la señal de concesión al dispositivo que le sigue en la cadena. De esta forma, los dispositivos que están eléctricamente más cerca del árbitro tienen prioridades mayores que los que están más alejados.

En muchos casos, el criterio de ordenamiento absoluto de prioridades no es el más adecuado, por lo que se usan varias líneas de pedido y concesión del bus (figura 8.6b). Las líneas de pedido de menor numeración tienen prioridades más altas que las líneas identificadas con números más altos. Con el objeto de mejorar la prioridad de un dispositivo que se encuentra alejado del árbitro, se lo asigna a una línea de pedido de menor numeración. Las prioridades correspondientes a un grupo de dispositivos conectados a la misma línea de pedido se asignan en función de su proximidad eléctrica al árbitro.

Llevando este modelo a su extremo, cada dispositivo puede tener su propio par de líneas de pedido y concesión del bus, como lo muestra la figura 8.6c. Este modelo completamente centralizado es el más potente desde el punto de vista lógico, pero, desde un punto de vista más práctico, es la menos escalable de todas las soluciones. El costo más significativo es la necesidad de líneas adicionales (elemento preciado) sobre el bus.

En la cuarta solución se presenta un esquema de arbitraje descentralizado, el que se muestra en la figura 8.6d. Nótese la falta de un árbitro principal. Un dispositivo que necesita utilizar el bus y convertirse en maestro activa primero su línea de pedido de uso del bus, y luego verifica si el bus está ocupado. Si corrobora que ningún otro dispositivo está utilizando el bus, el solicitante le envía un cero al dispositivo que le sigue en la cadena en sentido creciente, activa la línea de ocupado y deshabilita su línea de pedido. Cuando un dispositivo recibe una señal de concesión y no ha requerido el uso del bus, simplemente propaga la línea de concesión hacia el próximo dispositivo.

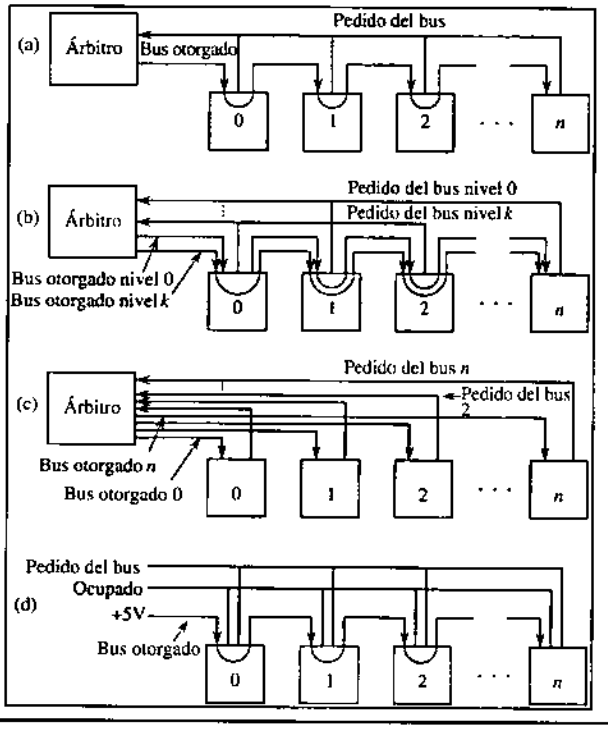


Figura 8.6 • (a) Arbitraje de bus simple y centralizado; (b) arbitraje centralizado con niveles de prioridad; (c) arbitraje completamente centralizado; (d) arbitraje descentralizado (adaptado de Tanenbaum, 2000).

La operación de arbitraje debe ser rápida, razón por la cual los esquemas centralizados solo funcionan bien cuando el número de dispositivos es bajo (no más allá de ocho). Para un número mayor de dispositivos es más apropiado el esquema descentralizado.

Dado un sistema que hace uso de alguno de estos esquemas de arbitraje, supóngase una configuración en la que se utilizan n dispositivos, y en la que se elimina el dispositivo m, con $m<n$. Dado que cada línea de pedido de bus está directamente conectada a todos los dispositivos de un grupo, y dado que la línea de concesión se transfiere a lo largo de todos los dispositivos del grupo, un pedido proveniente de un dispositivo cuyo índice es mayor (prioridad menor) que m no podrá ver nunca la línea de concesión activada, lo que puede ocasionar una situación de desastre en el sistema. Este problema puede ser difícil de identificar, por cuanto el sistema puede llegar a funcionar indefinidamente hasta que se produzca el acceso a ese dispositivo de baja prioridad (y alta identificación).

Cuando se elimina un dispositivo, los demás dispositivos de prioridades menores deben ser reubicados para completar el lugar vacío, salvo que se utilice un elemento como puente para mantener la continuidad de la línea de concesión. Los dispositivos rápidos (como los controladores de disco) deben tener prioridades mayores que los dispositivos lentos (como terminales) y, por consiguiente, deben ubicarse cerca del árbitro en un esquema centralizado, o bien, cerca del comienzo de la línea de concesión si el esquema es descentralizado. Esta solución resulta imperfecta ya que brinda la posibilidad de dejar

huecos en el bus o de desordenar la distribución de dispositivos. Por tal razón, hoy es frecuente encontrar que cada dispositivo tenga un trayecto separado hacia el árbitro.

8.2 Arquitecturas de bus basadas en puentes

Desde un punto de vista lógico, todos los componentes del sistema de la sección anterior se conectan directamente al bus del sistema. Desde el punto de vista operativo, esta solución suele traer problemas sobre el bus porque no pueden existir transferencias simultáneas entre los diversos dispositivos. Debido a que todos los dispositivos siempre están conectados al bus en forma simultánea, puede surgir en algún momento la necesidad de efectuar transferencias independientes. Por ejemplo, un elemento gráfico puede estar redibujando una pantalla de video a la vez que desde la memoria principal se recupera una línea de memoria *cache*, y esto en forma simultánea con una transferencia de entrada-salida sobre una red.

Las distintas transferencias suelen separarse sobre buses diferentes a través del uso de **puentes**. La figura 8.7 ilustra la implementación de puentes por medio de los procesadores Pentium II Xeon de Intel. En la parte superior del diagrama se observan dos procesadores Pentium II, estructurados en una configuración de **multiprocesador simétrico** (SMP). El sistema operativo realiza el balance de cargas por medio de la selección de un procesador sobre el otro en el momento de la asignación de tareas (este esquema es diferente al del procesamiento paralelo, analizado en el capítulo 10, en el que varios procesadores trabajan sobre un mismo problema). Cada procesador Pentium II tiene un "bus posterior" (*backside bus*) hacia su propia memoria *cache*, de 3.200 Mbytes/s (8 bytes x 400 MHz), con lo que se segrega el tráfico de la memoria *cache* del resto del tráfico sobre el bus.

Por debajo de la parte superior del diagrama, los dos procesadores Pentium II convergen sobre el bus del sistema (a veces conocido como bus frontal, o "*frontside bus*"). El bus del sistema tiene una capacidad de 32 bits y realiza transferencias en ambos flancos, creciente y decreciente, de la señal de reloj del bus, de 100 MHz, dando como resultado un ancho de banda total disponible de 4 bytes x 2 flancos x 100 MHz = 800 Mbytes/s, que se comparte entre los procesadores.

En el centro del diagrama se encuentra el *host bridge* Intel 440GX AGPset, el que conecta el bus del sistema a los restantes buses. El *host bridge* actúa como un intermediario entre el bus del sistema, la memoria principal, el procesador gráfico y una jerarquía de diferentes buses. A la derecha del *host bridge* se observa la memoria principal (una memoria dinámica, sincrónica), que se conecta al mismo por un bus de 800 Mbytes/s.

En este ejemplo particular, el *host bridge* provee al procesador gráfico de un bus separado conocido como AGP (*Advanced Graphic Port*)* sobre un bus de 533 Mbytes/s. La

* *N. de T.*: Las siglas mencionadas en esta sección responden a distintos formatos de interconexión utilizados habitualmente, y vastamente conocidos a través de sus siglas inglesas, por lo que no parece apropiado traducirlas al español.

composición (*rendering*) de gráficos (esto es, rellenar un objeto con colores) habitualmente necesita información de textura que es demasiado voluminosa como para ser colocada en una placa gráfica. La interfaz AGP permite generar un trayecto de alta velocidad entre el procesador gráfico y la memoria principal, en la cual pueden almacenarse adecuadamente los mapas de textura.

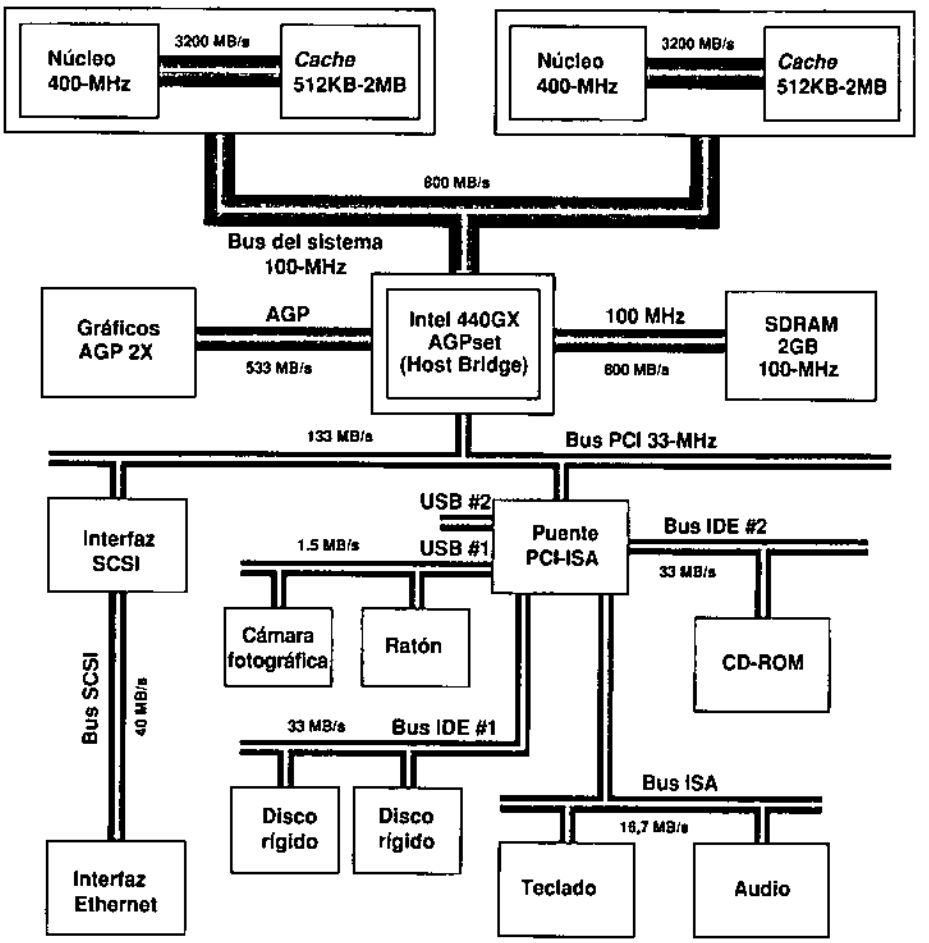


Figura 8.7 • Interconexión a través de puentes de dos procesadores Pentium II Xeon (fuente: http://www.intel.com).

Por debajo del *host bridge* hay un bus para interconexión de los dispositivos periféricos (PCI, *Peripheral Component Interconect*), de 33 MHz que conecta el *host bridge* con los restantes buses. Sobre el bus PCI hay otros componentes conectados, como un controlador tipo SCSI (*Small Computer System Interface*), correspondiente a otro tipo de bus, que, en este diagrama, contiene una placa de interfaz a red Ethernet. Antes de la introducción del sistema AGP, las placas gráficas se colocaban directamente sobre el bus PCI, lo que generaba un cuello de botella para el resto del tráfico sobre el bus.

Sobre el bus PCI aparece un puente de PCI a ISA, el que ofrece vinculación entre dos buses USB (*Universal Serial Bus*) de 1,5 Mbytes/s, dos buses IDE (*Integrated Drive Electronics*) de 33 Mbytes/s, y un bus ISA (*Industry Standard Arquitecture*) de 16,7 Mbytes/s. Los buses IDE se utilizan, generalmente, para la conexión de unidades de disco; el bus ISA se emplea para dispositivos de velocidad moderada, como impresoras y módems con voz, y los buses USB, para dispositivos lentos, como ratones y cámaras fotográficas digitales.

8.3 Metodologías de comunicación

Los sistemas de computación realizan una gran cantidad de tareas que requieren distintos esquemas de comunicación. La CPU debe comunicarse con la memoria y con una amplia gama de dispositivos de entrada-salida, desde algunos dispositivos extremadamente lentos, como teclados, hasta dispositivos de alta velocidad, como las interfaces para unidades de disco y para redes. Pueden existir múltiples procesadores que se comuniquen entre ellos, directamente o a través de una memoria compartida, como la configuración Pentium II Xeon descripta en la sección anterior.

Tres de los métodos utilizados para la administración de entrada-salida son los que se conocen como **entrada-salida programada** (también conocida como entrada-salida por encuesta [*polling*]), **entrada-salida por interrupciones** y **acceso directo a memoria** (DMA, *Direct Memory Access*).

8.3.1 Entrada-salida programada

Considérese la lectura de un bloque desde el disco. En la técnica de entrada-salida programada, la CPU interroga a cada dispositivo para ver si requiere servicio. Si se considera una analogía con un restaurante, el caso sería el del camarero que se acerca al comensal para ver si está listo para hacer un pedido.

Las operaciones que se realizan cuando se administra la entrada-salida por medio de un programa se ilustran en el diagrama de flujo de la figura 8.8. La CPU verifica primero el estado del disco por medio de la lectura de un registro especial al que puede acceder en el espacio de memoria o por medio de una instrucción especial de entrada-salida, si esta fuera la solución adoptada por la arquitectura del procesador para la implementación del manejo de entrada-salida. Si el disco no estuviese listo para ser leído o escrito, el proceso queda en un lazo cerrado verificando el estado en forma continua hasta tanto el disco esté listo, en un proceso de **espera por dispositivo ocupado**. Cuando el disco finalmente se encuentre disponible, se producirá la transferencia de datos entre el disco y la CPU.

Luego de completada la transferencia, la CPU verifica la existencia de algún otro pedido de comunicación con el disco. Si lo hubiera, el proceso se repite; en caso contrario, la CPU continúa con otras tareas.

En la administración de entrada-salida por programa, la CPU pierde tiempo al interrogar a los dispositivos. Otro problema tiene que ver con que los dispositivos de alta prioridad no se verifican hasta tanto la CPU no completa la tarea de entrada-salida en curso, la que puede ser de baja prioridad. No obstante, dado que la implementación de un sistema de administración programada de los procesos de entrada-salida se puede concretar en forma sencilla, presenta ventajas en algunas aplicaciones.

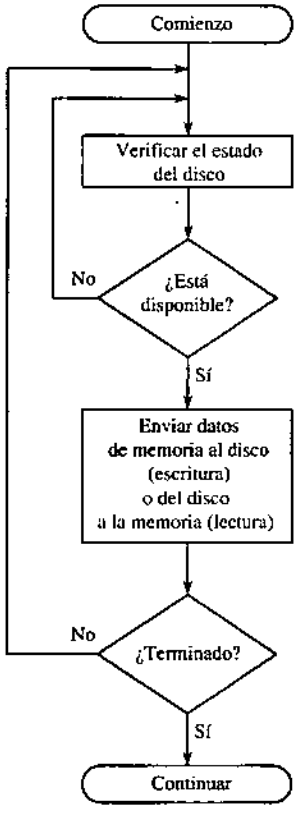


Figura 8.8 • Diagrama de flujo para una transferencia de disco mediante administración programada.

8.3.2 Entrada-salida administrada por interrupciones

Cuando los procesos de entrada-salida se administran por interrupciones, la CPU no accede a un dispositivo hasta que el mismo no requiera atención, por lo que no existen las esperas por dispositivo ocupado, como en el caso anterior. En este tipo de administración, el dispositivo requiere atención a través de una línea especial de pedido de interrupción que accede directamente a la CPU. En el ejemplo anterior del restaurante, el comensal golpea discretamente un cubierto contra una copa, llamando la atención del camarero cuando lo necesita.

El diagrama de flujo para la atención de entrada-salida por medio de interrupciones se muestra en la figura 8.9. La CPU emite un pedido al disco para proceder a una lectura o escritura, e inmediatamente retoma la ejecución de algún otro proceso. En algún momento posterior, cuando el disco se encuentra disponible, interrumpe a la CPU, la que invoca a la **rutina de atención de interrupciones** correspondiente a la atención del disco, tras lo cual vuelve a su funcionamiento normal cuando la rutina de atención de interrupciones completa su tarea. La rutina de atención de interrupciones es similar en estructura al procedimiento presentado en el capítulo 4, con una salvedad, consistente en que las interrupciones se producen en forma asincrónica con el programa que la CPU se encuentra ejecutando: las interrupciones pueden presentarse en cualquier momento durante la ejecución de un proceso.

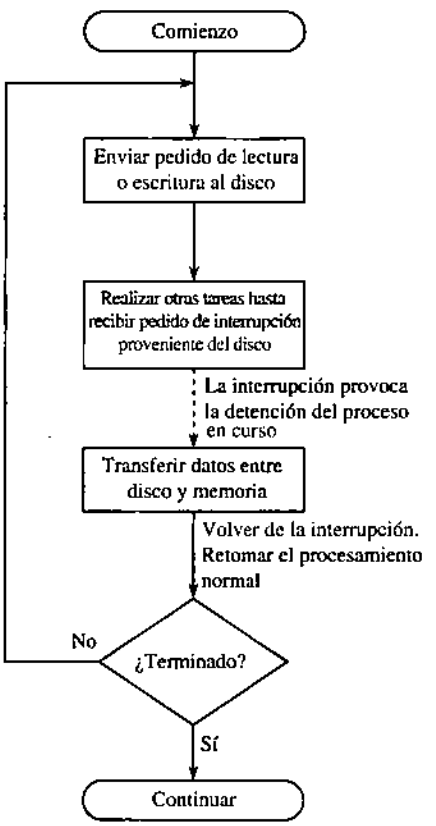


Figura 8.9 • Diagrama de flujo para una transferencia de disco mediante administración por interrupciones.

En ciertas ocasiones, suele requerirse que el proceso que la CPU está ejecutando no sea interrumpido debido, por ejemplo, a que se trata de alguna tarea crítica. Por esta causa, los juegos de instrucciones de los distintos procesadores incluyen instrucciones que permiten habilitar o inhibir la atención de las interrupciones bajo control del programa. (El mozo a veces puede ignorar al comensal.) El hecho de que la CPU acepte o no los pedidos de interrupción que recibe está determinado por el estado de la bandera de interrupciones (IF, *In-*

terrupt Flag) que forma parte del registro de estado del procesador. Más aún, en muchos sistemas se asignan prioridades a las interrupciones, las que son administradas desde la CPU o desde un **controlador de interrupciones** (PIC, *peripheral interrupt controller*). (El camarero puede atender primero a la mesa principal.) En el nivel más prioritario de la mayoría de los sistemas, existe una **interrupción no enmascarable** (NMI, *non maskable interrupt*) que, como su nombre lo indica, no puede ser inhabilitada. (El mozo atenderá siempre la alarma contra incendios.) La interrupción no enmascarable se utiliza tanto para la administración de eventos potencialmente catastróficos, por ejemplo una falla de alimentación, como para atender eventos más comunes pero no interrumpibles, por ejemplo una actualización de los archivos de sistema.

En el momento en que se atiende una interrupción, el procesador automáticamente coloca en la pila los registros de estado y contador de programa (`%psr` y `%pc` en el caso de ARC), tras lo cual se carga el contador de programa con la dirección de la rutina de atención de interrupciones que corresponda. El registro de estado se coloca en la pila debido a que contiene la bandera de interrupción (IF),* y el procesador debe inhibir las interrupciones, al menos durante la ejecución de la primera instrucción de la rutina de atención de la interrupción (véase el problema 8.2). A continuación, se inicia la ejecución de la rutina de atención de la interrupción. Cuando esta rutina completa su ejecución, el procesador reinicia la ejecución del programa interrumpido.

La instrucción `jmpl` de ARC (véase el capítulo 4) no sirve para retomar la ejecución del programa interrumpido, dado que además de recuperar el contador de programa almacenado en la pila, el procesador también necesita recuperar su registro de estado. En vez de aquella se utiliza la instrucción `rett` (retorno de interrupciones), la que revierte el proceso de interrupciones, recuperando en los registros `%psr` y `%pc` sus valores anteriores a la interrupción. En la arquitectura ARC, `rett` tiene un formato aritmético con `op3 = 111001` y su campo `rd` sin utilizar (todos ceros).

8.3.3 Acceso directo a memoria

Si bien la administración de procesos de entrada-salida por medio de interrupciones libera al procesador hasta tanto un dispositivo requiera servicios, la CPU sigue siendo responsable por la transferencia de los datos. La figura 8.10 resalta el problema. Con el objeto de transferir un bloque de datos entre la memoria principal y la unidad de disco, ya sea mediante técnicas de entrada-salida programada o administrada por interrupciones, cada palabra viaja por el bus del sistema (o, en términos equivalentes, por el *host bridge*) dos veces: una, hacia la CPU, y otra, nuevamente a través del bus del sistema hacia su destino.

* *N. de T.*: El registro de estado se rescata en la pila, además, para asegurar que las interrupciones atendidas por el procesador resulten transparentes al programa interrumpido, lo que significa que la primera instrucción del programa principal que se ejecute luego de la interrupción recupere la estructura de registros tal como si no hubiese habido interrupción alguna.

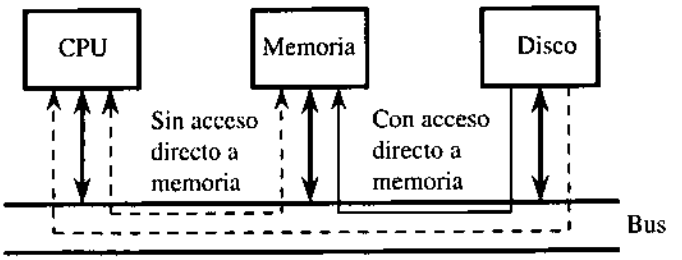


Figura 8.10 • El acceso directo a memoria desde la unidad de disco no requiere la intervención de la CPU.

Un dispositivo con acceso directo a memoria (DMA) puede transferir datos directamente hacia y desde la memoria en lugar de requerir el uso de la CPU como elemento intermediario, con lo que permite liberar parte de la congestión sobre el bus del sistema. Continuando con la analogía del restaurante, el camarero sirve a todos los comensales de una mesa antes de atender a cualquier persona en otra mesa. La atención de los pedidos de acceso directo a memoria suele ser administrada por un controlador de DMA, el que por su propia naturaleza es un procesador dedicado cuya especialidad es la de transferir datos directamente entre los dispositivos de entrada-salida y la memoria. La mayor parte de los controladores de DMA pueden programarse también para mover bloques de datos entre memoria y memoria. Por consiguiente, el dispositivo con acceso directo a memoria toma a su cargo la tarea de la CPU durante la transferencia. Al iniciar la transferencia, la CPU programa al dispositivo que realizará el acceso directo a memoria con la dirección de comienzo en memoria principal, la dirección de comienzo en el dispositivo y la longitud del bloque a transferir.

La figura 8.11 ilustra el proceso de acceso directo a memoria para una transferencia con un disco. La CPU inicializa el dispositivo y le indica que comience la transferencia. Mientras la misma se lleva a cabo, la CPU continúa con la ejecución de otro proceso.* Cuando se completa la transferencia de acceso directo a memoria, el dispositivo le informa a la CPU por medio de una interrupción. Todo sistema que puede implementar acceso directo a memoria también puede implementar interrupciones.

Si el dispositivo con acceso directo a memoria requiere transferir un bloque de datos de gran tamaño sin liberar el bus, la CPU puede quedarse sin instrucciones ni datos, con lo que deberá detener su trabajo hasta que se haya completado la transferencia de acceso directo a memoria. Con el objeto de paliar el problema, los controladores de acceso directo a memoria usan normalmente un esquema de "robo de ciclos" (*cycle stealing*). En el acceso directo a memoria por **robo de ciclos**, el controlador toma el uso del bus, transfiere una única palabra y vuelve a liberar el bus. Esto permite que otros dispositivos, y en particular la CPU, compartan el uso del bus durante las transferencias de acceso directo a memoria. En la analogía del restaurante, un comensal puede pedir la cuenta mientras el camarero sirve otra mesa.

* *N. de T.*: En tanto y en cuanto no requiera el uso del bus.

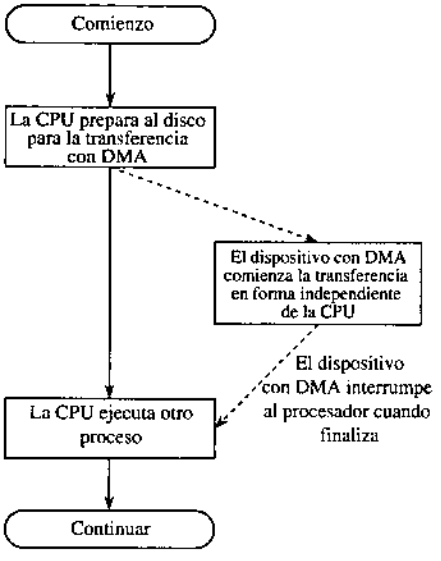


Figura 8.11 • Diagrama de flujo para una transferencia con un disco a través del acceso directo a memoria.

8.4 Estudio de un caso: comunicaciones en la arquitectura Pentium de Intel

La familia de procesadores Pentium de Intel es la implementación actualizada de su venerable familia x86, la que nace con el 8086 lanzado al mercado en 1978. Pentium es una familia de procesadores que incluye versiones que enfatizan la alta velocidad, los ambientes multiprocesadores, gráficos, bajo consumo, etc. En esta sección se analizan las características del bus del sistema Pentium, que conecta al procesador con el *host bridge* (véase la figura 8.2).

8.4.1 El reloj del sistema, el reloj del bus y las velocidades del bus

Es interesante observar que la frecuencia del reloj del sistema se establece como un múltiplo del reloj del bus. El procesador determina el valor del múltiplo cada vez que se lo reinicializa, de acuerdo con los valores presentes en varios de sus terminales. Los posibles valores del multiplicador varían a lo largo de los distintos miembros de la familia. Por ejemplo, en el caso del Pentium Pro, un elemento adaptado para aplicaciones de múltiples procesadores, los valores de los factores multiplicadores puden variar entre 2 y 3,5. Se vuelve a reiterar aquí que la necesidad de controlar el bus del sistema a una frecuencia menor que la de la CPU tiene que ver con que las operaciones internas de la CPU pueden realizarse a una velocidad mayor que las operaciones de acceso a memoria principal. Una frecuencia habitual para el bus de los sistemas Pentium es de 66 MHz.

8.4.2 Direcciones, datos, memoria y entrada-salida

El bus del sistema tiene efectivamente 32 líneas de dirección, por lo que puede acceder hasta a 4 GB de memoria principal. El bus de datos tiene 64 bits; esto hace posible que el procesador transfiera una palabra de 8 bytes en un solo ciclo de bus. (Las palabras de los procesadores Intel x86 son de 16 bits.) El término "efectivamente" se utiliza porque, en realidad, el procesador Pentium decodifica las tres líneas menos significativas de direcciones, A_2-A_0, y las convierte en ocho líneas de "habilitación de bytes", *BE0#-BE7#*, antes de colocarlas sobre el bus del sistema.[1] Los valores que adoptan estas ocho líneas identifican al byte, la palabra de 16 bits, la palabra doble de 32 bits o la palabra cuádruple de 64 bits que se transferirá desde la dirección de base especificada por A_{31}-A_3.

8.4.3 Las palabras de datos no requieren alineación forzada

Los valores de datos requieren la así llamada alineación no forzada; lo que significa que las palabras deben estar alineadas en dirección par, sean simples, dobles o cuádruples, con el objeto de lograr la máxima eficiencia, aunque el procesador puede tolerar elementos de datos desalineados. La penalización por el acceso a palabras desalineadas puede equivaler a dos ciclos de bus, los que se hacen necesarios para acceder a cada una de las mitades de los datos.[2]

Como concesión a los espacios limitados de direcciones de los primeros miembros de la familia, todos los procesadores Intel tienen espacios de direcciones separados para los accesos a memoria y a la entrada-salida. El espacio de memoria a seleccionar queda determinado por la línea de bus M/IO#. En esta línea, el nivel alto selecciona el espacio de direcciones de memoria de 4 GB, y el nivel bajo especifica el espacio de entrada salida, al que se tiene acceso con dos códigos de operación específicos, IN y OUT. Es responsabilidad de cada uno de los dispositivos conectados sobre el bus la verificación de la línea de M/IO# al comienzo de cada ciclo de reloj, con el objeto de determinar cuál es el espacio de direcciones al que se refiere el ciclo de bus, si el de memoria o el de entrada-salida. La figura 8.12 ilustra gráficamente estos dos espacios. Las direcciones en el espacio de entrada-salida están limitadas a 16 bits en la familia x86, permitiendo el acceso de hasta 64 K locaciones de entrada-salida.

8.4.4 Ciclos de bus en la familia Pentium

El procesador Pentium tiene un total de 18 ciclos de bus diferentes, con el objeto de satisfacer distintas necesidades. Estos incluyen las operaciones normales de lectura y escritura de memoria, la operación de retención del bus (requerida para permitir que otros

1. El símbolo "#" es la notación que utiliza Intel para indicar una línea del bus que se activa en estado bajo.
2. Muchos sistemas requieren la así llamada alineación forzada. No se admiten palabras desalineadas y su detección puede generar una falla o excepción del procesador.

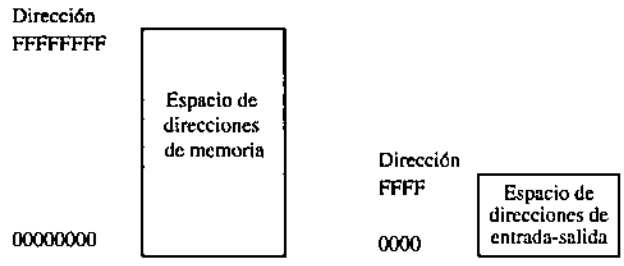


Figura 8.12 • Espacios de direcciones de memoria y de entrada-salida en la arquitectura Intel.

dispositivos se conviertan en maestros del bus), un ciclo de reconocimiento de interrupciones, varios ciclos de acceso a memoria *cache* en forma de bloques, y un conjunto de varios otros tipos de ciclos de bus de propósitos específicos. En este análisis se examinan los ciclos de bus de lectura y escritura, el ciclo de lectura por bloques, en el que se puede transferir un bloque de datos, y el ciclo de retención de bus y su reconocimiento, el que suele ser usado por aquellos dispositivos que pretenden convertirse en maestros de bus.

8.4.5 Ciclos de bus de lectura y escritura de memoria

La figura 8.13 ilustra los ciclos de lectura y escritura "normales". Por convención, los estados del bus Intel se conocen como "estados T", donde cada uno de estos estados corresponde a un ciclo de reloj. La figura muestra tres estados T: T1, T2 y Ti, donde Ti es el estado "inactivo", que se produce cuando el bus no está ocupado en actividad especial alguna y no hay pedidos pendientes para usar el bus. Nótese que un "#" después del nombre de una señal indica que la señal es activa en su estado bajo, de acuerdo con las convenciones Intel. De la misma forma, se utilizarán letras mayúsculas para representar las señales específicas de Intel.

Ambos ciclos de lectura y escritura requieren un mínimo de dos ciclos de reloj, T1 y T2:

- La CPU indica el comienzo de cada nuevo ciclo de bus mediante la validación de la señal ADS# (*Address Status*). Esta señal define el comienzo de un nuevo ciclo de bus y, además, le avisa a la memoria que la dirección sobre el bus de direcciones ADDR es válida. Nótese la transición de ADDR de inválida a válida cuando ADS# está en su valor activo.
- La desactivación de la señal de carga de la memoria *cache*, CACHE#, indica que el ciclo se conformará con una única lectura o escritura, a diferencia de los ciclos de lectura o escritura en bloques, que se detallarán más adelante.
- Durante un ciclo de lectura, la CPU valida la señal de lectura W/R#, simultáneamente con la aserción de ADS#. Esto le indica a la memoria principal que deberá memorizar la dirección y leer el contenido de la dirección especificada.
- En la lectura, la memoria principal valida la señal BRDY# (*Burst Ready*) a la vez que coloca los datos, DATA, sobre el bus, indicando la existencia de un dato válido

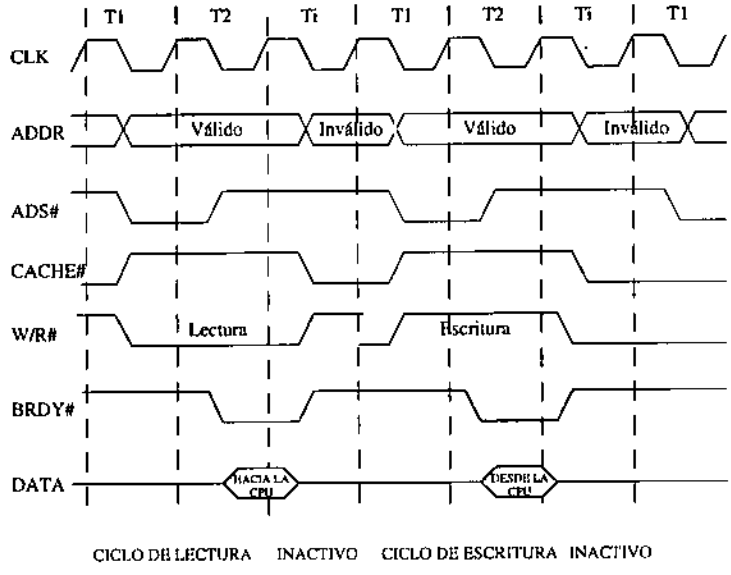


Figura 8.13 • Los ciclos normales de lectura y escritura del procesador Pentium Intel.

sobre los terminales de datos. El procesador usa BRDY# para memorizar los valores de los datos.

- Dado que CACHE# está deshabilitada, la validación de una única señal BRDY# implica el final de ciclo de bus.
- En el ciclo de escritura, la memoria principal valida BRDY# cuando está lista para aceptar los datos que la CPU colocó sobre el bus. La señal BRDY# actúa como señal de diálogo entre memoria y CPU.
- Si la memoria es demasiado lenta para aceptar o entregar datos dentro del límite de dos ciclos de reloj, puede insertar estados de espera simplemente no validando BRDY# hasta que esté lista para responder.

8.4.6 El ciclo de bus de lectura por bloques (burst)

Dada la necesidad crítica de proveer a la CPU con instrucciones y datos desde una memoria que es naturalmente más lenta que la CPU, Intel desarrolló los ciclos de lectura y escritura por bloques. Estos ciclos leen y escriben cuatro palabras de ocho bytes cada una en un bloque, a partir de direcciones consecutivas. La figura 8.14 ilustra el ciclo de lectura por bloques del procesador Pentium.

El procesador inicia un ciclo de lectura por bloques colocando una dirección sobre las líneas de dirección y validando ADS# de la misma forma que en el caso anterior. Con el objeto de indicar la iniciación de un ciclo de lectura de un bloque, el procesador valida la línea CACHE#, como respuesta a lo cual la memoria valida BRDY# y coloca una secuencia de cuatro palabras de 64 bits, una por cada ciclo de reloj, manteniendo BRDY# válida hasta que se complete la transferencia íntegra de las mismas.

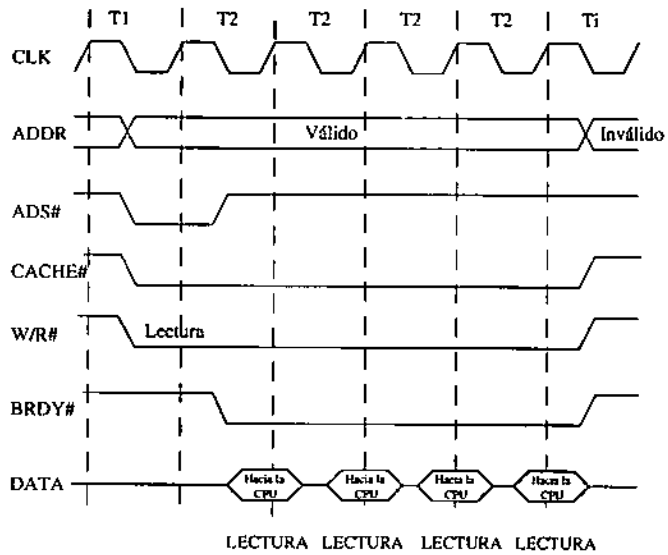


Figura 8.14 • El ciclo de lectura por bloques en el procesador Pentium Intel.

Existe un ciclo similar para escrituras en bloque. Asimismo, hay un mecanismo para enfrentar a las memorias más lentas, en el cual la frecuencia de transferencia en bloques se modifica a una transferencia por cada dos ciclos de reloj.

8.4.7 Retención del bus para admitir pedidos de bus por parte de otro maestro

Existen dos señales de bus disponibles para los dispositivos que requieren ser maestros del mismo: una señal de retención (HOLD) y otra de reconocimiento (HLDA). La figura 8.15 ilustra la forma de funcionamiento de este tipo de transacción. Con respecto a la figura, se supone que el procesador está en el medio de un ciclo de lectura cuando recibe la señal HOLD de pedido. El procesador completa el ciclo corriente de lectura e inserta dos ciclos de inactividad Ti. Durante el flanco de caída del segundo ciclo Ti, el procesador coloca todas sus líneas en estado flotante (alta impedancia) y activa la señal HLDA, la que se mantiene activa durante dos ciclos de reloj. Al final del segundo ciclo de reloj, el dispositivo que había solicitado el HOLD pasa a tener el control del bus, pudiendo iniciar una nueva operación en el siguiente ciclo de bus, tal como se observa a la derecha de la figura. Normalmente, cualquiera sea la complejidad del sistema, podrá existir un circuito controlador de bus independiente que sirva como mediador entre los distintos dispositivos que desean ser maestros del bus.

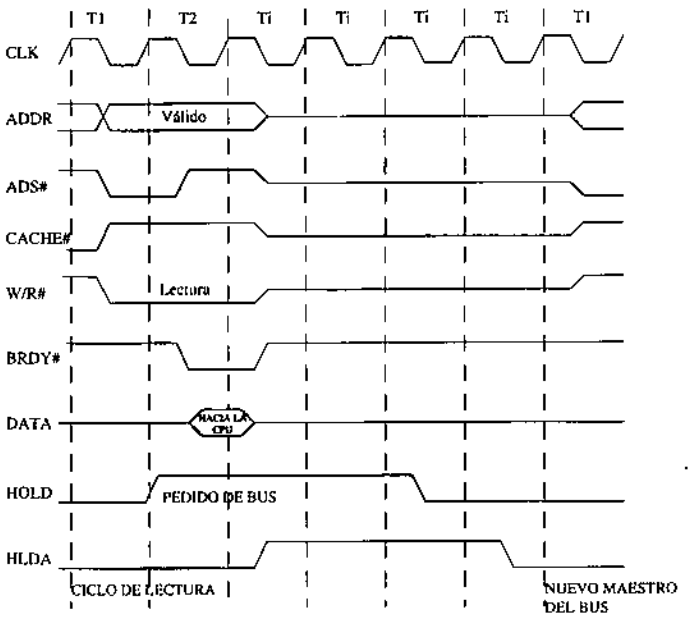


Figura 8.15 • El ciclo de bus de pedido de retención y reconocimiento de retención en la arquitectura Pentium Intel.

8.4.8 Velocidades de transferencia de datos

Se pretende calcular las velocidades de transferencia de datos en los ciclos de bus de lectura y de lectura por bloques. En el primer caso, se transfieren 8 bytes en dos ciclos de reloj. Si la frecuencia del reloj del bus es de 66 MHz, se obtiene una velocidad máxima de transferencia de $8/2 \times 66 \times 10^6 = 264$ Mbytes/s. En el modo de transferencia por bloques, esta velocidad se incrementa, dado que se transfieren cuatro bloques de 8 bytes cada uno en cinco ciclos de reloj, a una velocidad de $32/5 \times 66 \times 10^6 = 422$ Mbytes/s. (En la literatura de Intel se hace mención a cuatro ciclos de reloj en vez de cinco, lo que lleva a una velocidad de transferencia de 528 Mbytes/seg. El valor correcto queda a elección del lector.)

A una velocidad de 422 Mbytes/s, con un multiplicador de reloj de 3,5 veces, la velocidad de transferencia hacia la CPU es de $(422 \times 10^6)/(3{,}5 \times 66 \times 10^6)$, lo que equivale, aproximadamente, a 2 bytes por ciclo de reloj. Aun en condiciones ideales u óptimas, la velocidad de transferencia de información hacia la CPU apenas si alcanza para satisfacer sus necesidades. En el caso de una instrucción de salto o alguna otra interrupción en la actividad de la memoria, la CPU se quedaría sin instrucciones ni datos.

La arquitectura Pentium Intel es típica de los procesadores modernos. Tiene una cantidad de ciclos de bus especializados que soportan multiprocesamiento, transferencias de memoria *cache* y otras situaciones. Se remite al lector a la literatura propia de Intel (para más detalles véase la sección "Para lectura posterior", al final del capítulo).

8.5 Almacenamiento masivo

En el capítulo 7 se ha visto que la memoria de la computadora se organiza como un esquema de jerarquías, en la que el método más rápido para el almacenamiento de información (registros) es caro y de baja densidad, en tanto que los métodos más lentos de almacenamiento (discos, cintas, etc.) son económicos y de alta densidad. Los registros y las memorias de lectura-escritura requieren alimentación continua de energía eléctrica para mantener la información almacenada, en tanto que los **medios** magnéticos, como las cintas y los discos magnéticos, retienen la información aun después de habérseles quitado la alimentación eléctrica, lo que se denomina **persistencia indefinida**. Este tipo de almacenamiento se puede definir como **no volátil**. Existen muchos tipos de almacenamiento no volátil, de los cuales se describen aquí solo algunos de los más comunes. Para comenzar, se analiza una de las formas más dominantes: el **disco magnético**.

8.5.1 Discos magnéticos

Un disco magnético es un dispositivo para almacenamiento de información con una gran densidad de almacenamiento y un tiempo de acceso relativamente corto. Un disco magnético de **cabeza móvil** está compuesto por una pila de uno o más **platos** separados por algunos milímetros uno del otro y conectados a un **eje**, como lo ilustra la figura 8.16. Cada uno de los platos tiene dos **superficies** de aluminio o vidrio (el que se dilata menos que el aluminio cuando el disco toma temperatura), cubiertas de pequeñas partículas de material magnético, como óxido de hierro, las que provocan el color marrón de los distintos tipos de medios magnéticos, por ejemplo: los platos de discos, los discos flexibles (disquetes), las cintas de audio, etc. Se almacenan ceros y unos por medio de la magnetización de pequeñas áreas del material.

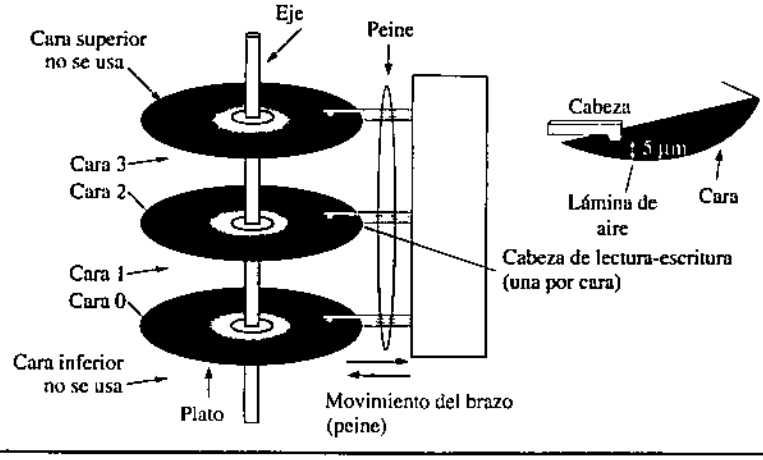


Figura 8.16 • Un disco magnético con tres platos.

Cada cara tiene asignada una **cabeza** única. En el ejemplo de la figura 8.16, de seis caras, las cabezas a utilizar son seis. La cara superior del plato superior, y la cara inferior del plato inferior normalmente se dejan sin utilizar en discos de múltiples platos, debido a que son más sensibles a la contaminación que las caras internas. Las cabezas se fijan a un **brazo** o **peine** común, que se mueve en sentido radial, hacia adentro y hacia fuera, para alcanzar distintas porciones de las superficies.

En un disco magnético de este tipo (disco rígido) los platos giran a una velocidad constante, que generalmente se encuentra en el rango de 3.600 a 10.000 rpm. Las cabezas permiten la lectura y escritura de la información por medio de la magnetización del material que pasa por debajo de la cabeza, en el caso de la escritura, o por medio de la verificación del campo magnético en el caso de la lectura. En cada momento se permite el funcionamiento de una sola cabeza, sea para lectura o para escritura, por lo que la información se almacena en serie aun cuando las cabezas podrían usarse en principio para la lectura o escritura de varios bits en paralelo. Una razón por la que no se utiliza la operación en modo paralelo es la posibilidad cierta de desalineación de las cabezas, lo que corrompe la lectura o escritura de la información. Una sola superficie es menos sensible a la alineación de la cabeza correspondiente, debido a que siempre se conoce fehacientemente la posición de la cabeza con respecto a marcas de referencia que se encuentran en el disco.

Codificación de la información

Durante la lectura de un disco, solo se detectan las transiciones entre áreas magnetizadas, por lo que las series de ceros o de unos pueden no detectarse correctamente, a menos que se incorpore un método de codificación que incluya información temporal dentro de los datos de modo de permitir identificar las separaciones entre los sucesivos bits. Un método de codificación que intenta resolver este problema es el llamado **formato Manchester**; otro método es el de la **modulación en frecuencia modificada** (MFM, *modified frequency modulation*). Con el objeto de comparar ambos métodos, la figura 8.17a ilustra la secuencia ASCII correspondiente a la letra "F", codificada en el formato de **no retorno a cero** (NRZ, *non return to zero*), formato utilizado en las transferencias internas de la CPU. La figura 8.17b muestra el mismo carácter, ahora codificado en formato Manchester. En la codificación Manchester hay una transición entre los niveles alto y bajo en cada bit, por lo que se obtiene una transición por cada tiempo bit.* Una transición desde un nivel bajo a un nivel alto indica un uno, en tanto que una transición desde un nivel alto a un nivel bajo representa un cero. Estas transiciones se utilizan para recuperar la información de temporización.

Una única cara contiene varios cientos de **pistas** concéntricas, las que a su vez están compuestas por **sectores**, cuyo tamaño típico es de 512 bytes, almacenados en

* *N. de T.*: Suele denominarse "tiempo bit", en una transmisión de datos en serie, al tiempo durante el cual cada uno de los bits de una palabra se mantiene vigente sobre la línea de transmisión.

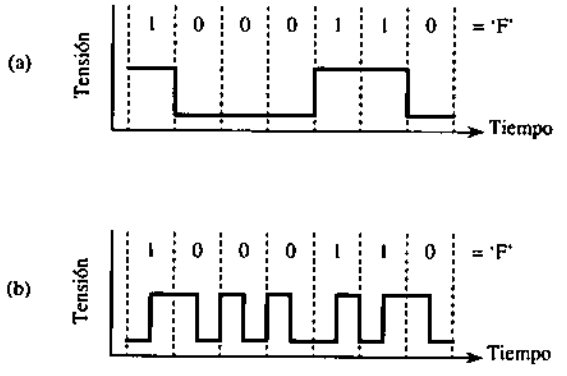


Figura 8.17 • (a) Codificación sin retorno a cero de la letra F representada en ASCII. (b) Codificación Manchester del mismo carácter.

serie, tal como se muestra en la figura 8.18. Los sectores se encuentran separados por **espacios intersectoriales**, en tanto que las pistas se encuentran separadas por un **espacio entre pistas**, los que simplifican el posicionamiento de la cabeza. El conjunto de pistas congruentes en todas las superficies forma un **cilindro**. Por ejemplo, las pistas 0 de cada una de las superficies 0, 1, 2, 3, 4 y 5 de la figura 8.16 constituyen el cilindro 0. La cantidad de bytes por sector generalmente se mantiene invariable a lo largo de todo el plato.

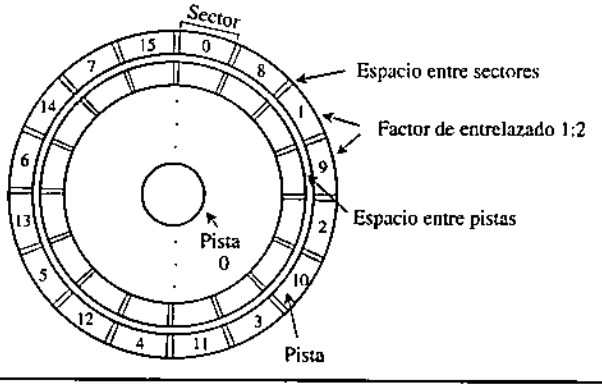


Figura 8.18 • Organización de un disco con factor de entrelazado de 1:2.

En los discos modernos, la cantidad de sectores por pista puede variar por **zonas**; una zona es el conjunto de pistas que tienen el mismo número de sectores por pista. Las zonas cercanas al centro del plato, en las que los bits están muy próximos unos a otros, tienen menor cantidad de sectores, en tanto que en las pistas externas, de menor densidad de bits, se pueden tener más sectores por pista. Esta técnica, que permite incrementar la capacidad de un disco, se conoce como **grabación por zonas**.

Capacidades y velocidades de los discos

Cuando la estructura de un disco está constituida por una única zona, su capacidad de almacenamiento C puede calcularse a partir de la cantidad N de bytes por sector, de la cantidad S de sectores por pista, del número T de pistas por superficie y de la cantidad P de superficies que almacenan información, lo que da

$$C = N \times S \times T \times P$$

Una unidad de disco de alta capacidad puede tener $N = 512$ bytes/sector, $S = 1000$ sectores/pista, $T = 5000$ pistas/cara y $P = 8$ platos. La capacidad total de esta unidad de disco es $C = 512$ bytes/sector x 1000 sectores/pista x 5000 pistas/superficie x 8 platos x 2 superficies/plato = 38 GB.

La máxima velocidad de transferencia de información está controlada por tres factores: el tiempo necesario para mover la cabeza hasta la pista deseada, lo que se conoce como **tiempo de búsqueda** (*seek time*), el tiempo necesario hasta encontrar el sector deseado bajo las cabezas de lectura-escritura, conocido como **latencia de rotación**, y el tiempo necesario para transferir el sector desde el disco, una vez que el mismo se ha posicionado bajo las cabezas, que se conoce como **tiempo de transferencia**. Las transferencias hacia o desde el disco se llevan a cabo por sectores completos, siendo imposible la lectura o escritura parcial de sectores de datos.

El tiempo de búsqueda de las cabezas es el que más aporta al tiempo total de acceso a un disco. En general, los fabricantes especifican un tiempo de búsqueda promedio, que es aproximadamente el tiempo requerido para que la cabeza se desplace en un recorrido igual a la mitad del radio del disco. La justificación de esta definición es que resulta difícil conocer a priori cuál es la pista en la que se encuentra la información o dónde se encuentra ubicada la cabeza cuando se realiza el pedido de acceso a disco. Por lo tanto, se asume que la cabeza necesitará, en promedio, recorrer la mitad de la superficie antes de llegar a la pista que corresponde. En las unidades de disco actuales el tiempo medio de búsqueda está por debajo de los 10 ms.

Una vez que la cabeza se encuentra ubicada sobre la pista correcta, vuelve a ser difícil saber cuánto tiempo se necesitará para que el sector buscado aparezca bajo las cabezas. Por lo tanto, el tiempo medio de latencia de rotación se estima en la mitad de una revolución completa, que, según la velocidad de rotación, lleva entre 4 y 8 ms. El tiempo de transferencia del sector es el tiempo de una vuelta completa, dividida por la cantidad de sectores por pista. Si se requiere la transferencia de grandes cantidades de información, luego de haberse transferido una pista completa, la cabeza debe desplazarse a la pista siguiente. El parámetro que interesa en ese caso es el tiempo de acceso entre pistas, de aproximadamente 2 ms. (Nótese que el tiempo que necesita la cabeza para desplazarse a través de más de una pista es mucho menor que 2 ms por pista.) En relación con el tiempo de transferencia de un sector, se considera un parámetro importante a la **velocidad de transferencia**, me-

dida como la velocidad a la que fluye la información desde o hacia el disco una vez que se ha iniciado la operación de lectura o escritura. Este valor no coincide necesariamente con el tiempo de transferencia, debido al tiempo de inicialización requerido para posicionar la cabeza y generar los sincronismos en cada sector, y puede determinarse a través del producto de la velocidad de rotación del disco, medida en revoluciones/segundo, por la capacidad de la pista.

La máxima velocidad de transferencia calculada con los factores mencionados anteriormente puede no ser real en la práctica. El factor limitativo puede ser la velocidad del bus que interconecta a la unidad de disco con su interfaz, o el tiempo requerido por la CPU para transferir los datos entre el disco y la memoria principal. Por ejemplo, los discos que operan con la norma **SCSI** (*Small Computer Systems Interface*; interfaz para pequeños sistemas de computación) tienen una tasa de transferencia entre disco y computadora que se encuentra entre 5 y 40 Mbytes/s, la que puede ser menor que la velocidad de transferencia entre la cabeza y el *buffer* interno del disco. Las unidades de disco contienen *buffers* internos que colaboran para coordinar la velocidad del disco con la velocidad de transferencia desde la unidad de disco a la computadora con la que se comunica.

Las unidades de disco son mecanismos delicados

La energía de un campo magnético cae en función del cuadrado de la distancia al origen del campo, y por esta razón es importante que la cabeza del disco se desplace tan cerca de la superficie del mismo como sea posible. La distancia entre la cabeza y la superficie puede ser del orden de los 5 μ, a pesar de lo cual no es necesario que la ingeniería y el montaje de un disco se tengan que ajustar a tolerancias tan rígidas: el sistema de cabezas tiene un diseño aerodinámico que hace que el movimiento de rotación del disco genere un colchón de aire tal que mantenga una distancia entre las cabezas y la superficie. Si una partícula mayor que 5 μ, de las que se encuentran dentro del ámbito del disco, llega a introducirse entre el mecanismo de cabezas y la superficie, puede provocar un daño en las pistas por aterrizaje de la cabeza.

Las partículas de humo generadas por las cenizas de un cigarrillo tienen un tamaño de 10 μ o más, lo que exige que no se fume en ambientes en los que existen discos. Los discos se ensamblan habitualmente en unidades selladas, en ambientes asépticos; en consecuencia, no se introducen partículas extrañas durante el armado. Sin embargo, y desafortunadamente, los materiales que se utilizan durante el proceso de fabricación de la unidad (como los adhesivos) pueden sufrir deterioros con el paso del tiempo y liberar partículas lo suficientemente grandes como para producir un aterrizaje de cabezas. Por esta razón, los discos sellados (antiguamente conocidos como discos Winchester) contienen sistemas de filtrado que extraen las partículas generadas en el interior de la unidad y que evitan que las partículas de materia provenientes del ambiente externo penetren en la unidad de disco.

Discos flexibles (disquetes)

Los **discos flexibles**, conocidos como discos ***floppy*** o **disquetes**, están constituidos por una superficie de plástico flexible cubierta por un material magnético como el óxido de hierro. Aun cuando en muchos sistemas se utilice una sola cara de la superficie del disco, ambas caras se cubren con el mismo material para evitar deformaciones. El tiempo de acceso es, en general, mayor que el de un disco rígido debido a que los discos flexibles no pueden girar a tanta velocidad como los rígidos. La velocidad de rotación de un mecanismo típico de disco flexible es de solo 300 RPM, y puede variar en la medida en que la cabeza se mueve de una pista a otra, para optimizar las velocidades de transferencia de la información. Estas velocidades de rotación, bajas, implican que los tiempos de acceso de este tipo de discos están en el orden de los 250 a 300 ms, o sea, alrededor de 10 veces más lentos que los discos rígidos. Sus capacidades varían, pero pueden llegar hasta 1,44 MB.

Los discos flexibles son económicos debido a que pueden extraerse de la unidad que los gobierna y a su tamaño reducido. La cabeza funciona en contacto con la superficie, y si bien esto no significa un problema de daño por aterrizaje, sí desgasta la cabeza y el medio magnético. Por tal razón, este tipo de disco solo gira cuando se lo accede.

En los principios de la utilización de discos flexibles, estos eran envasados en envoltorios de material plástico flexible y delgado, lo que dio origen al nombre de "*floppies*". En la actualidad, las superficies flexibles se envasan en plástico rígido, y se las denomina "disquetes".

En los últimos años aparecieron varios mecanismos de almacenamiento de alta densidad, de aspecto similar a los discos flexibles, como el mecanismo Zip de la firma Iomega, que tiene una capacidad de 100 MB y tiempos de acceso que duplican el de los discos rígidos. Asimismo, el mecanismo Jaz de Iomega, de mayor capacidad, puede almacenar hasta 2 GB, con tiempos de acceso similares.

Entre otros mecanismos se encuentra el SuperDisk, desarrollado por Imation Corp, el que utiliza discos similares a los flexibles antes mencionados; tiene 120 MB de capacidad y puede leer y escribir los discos flexibles comunes de 1,44 MB.

Sistemas de archivos en discos

Un **archivo** es una colección de sectores vinculados para formar una única entidad lógica. Un archivo almacenado en un disco puede estar organizado de distintas formas. El método más eficiente es almacenarlo en sectores consecutivos de modo que el tiempo de búsqueda y la latencia de rotación se minimicen. Normalmente, un disco almacena más de un archivo, y en general es difícil predecir el tamaño máximo del archivo. En muchas aplicaciones suele ser apropiada la utilización de archivos de tamaño fijo. Por ejemplo, en el caso de las imágenes satelitales, todas las imágenes pueden tener, en un solo paso de muestreo, un mismo tamaño.

Un método alternativo para la organización de archivos es la asignación de sectores a demanda. Con este método, los archivos pueden crecer hasta tamaños arbitrarios, pero la lectura o la escritura pueden requerir demasiados movimientos de las cabezas. Cuando un sistema de discos se usa durante cierto tiempo, los archivos contenidos en él pueden quedar **fragmentados** (esto significa que los sectores que conforman el archivo se encuentran desparramados sobre las superficies del disco). Para solucionar este efecto, diversos fabricantes ofrecen programas optimizadores que permiten defragmentar el disco, reorganizándolo de modo tal que cada archivo quede almacenado nuevamente en sectores y pistas contiguas.

El **entrelazado** es otro de los temas vinculados con la organización del disco. Aun si la CPU y los circuitos de interfaz entre la unidad de disco y la CPU mantienen todos el ritmo con la velocidad interna del disco, puede haber algún problema oculto relacionado con el rendimiento del sistema. Luego de leer y almacenar un sector para su transferencia, se lo envía a la CPU. Si la CPU requiere en ese momento la lectura del siguiente sector contiguo al leído recién, puede ser muy tarde para leer dicho sector sin la necesidad de esperar una nueva revolución del disco. Si los sectores se encuentran entrelazados (por ejemplo, si un archivo se almacena en sectores alternados, como el 2, el 4, el 6, etc.), el tiempo requerido para que los sectores intermedios pasen por debajo de la cabeza puede ser suficiente como para preparar la transferencia siguiente. En este escenario, se necesitan dos o más revoluciones del disco para leer una pista completa, pero este tiempo es menor que el requerido para leer un sector en cada vuelta. Si no fuera suficiente con el tiempo de pasaje de un único sector por debajo de las cabezas, se podrá usar un factor de entrelazado mayor, como 1:3 o 1:4. En la figura 8.18, el ejemplo utiliza un factor de entrelazado de 1:2.

El sistema operativo es el responsable de la ubicación de los bloques (sectores) de un archivo creciente y de la lectura de los bloques que forman un archivo, por lo que necesita conocer dónde se encuentran los bloques. Se define como **bloque maestro de control** (MCB, *master control block*) a un sector reservado de un disco, en el que se lleva el control de la configuración de todo el resto del disco. Este bloque maestro ocupa generalmente la misma ubicación dentro de cualquier disco correspondiente a un determinado tipo de computadora, por ejemplo, la pista interna del disco. De esta forma, el sistema operativo no necesita adivinar el tamaño de un disco; solo debe leer el bloque maestro de control desde la pista interior del mismo.

La figura 8.19 ilustra una versión de un bloque maestro de control. No todos los sistemas almacenan toda esta información en el bloque maestro, pero la información debe estar contenida en algún lado, y parte de ella puede estar almacenada incluso como parte del mismo archivo. El bloque maestro de control contiene cuatro componentes principales. La sección de preámbulo especifica la información referida a las características físicas del disco, incluyendo la cantidad de superficies, la cantidad de sectores por superficie, etc. La sección de archivos ofrece una referencia cruzada entre los nombres de los archivos y la lista de sectores que los forman, además de informar sobre atributos de los archivos,

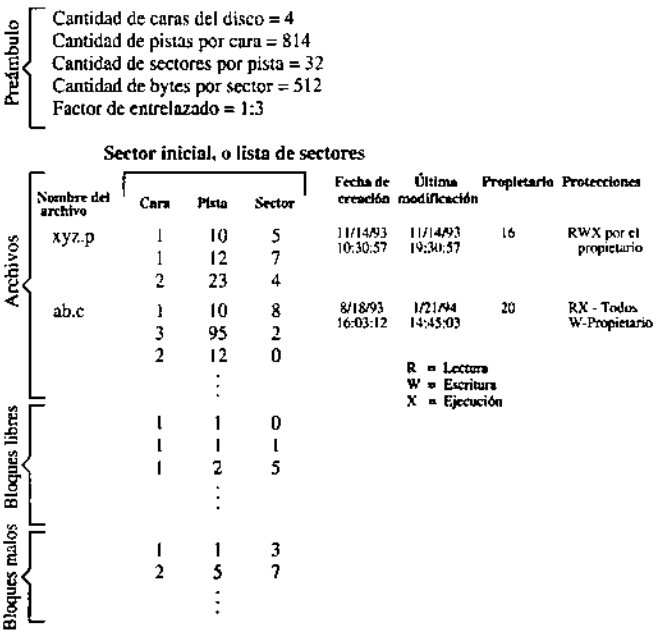


Figura 8.19 • Ejemplo simplificado de un bloque maestro de control.

tales como la fecha de creación, la fecha de la última modificación, la identificación del propietario y sus protecciones. Cuando el disco trabaja con archivos de tamaño fijo solo se requiere el dato del sector de comienzo; en caso contrario, debe mantenerse una lista de todos los sectores que constituyen el archivo.

La sección de bloques libres detalla las posiciones de aquellos bloques libres disponibles para archivos nuevos o de tamaño creciente. La sección de bloques malos señala las posiciones de aquellos bloques que se encuentran libres pero cuyos **algoritmos de verificación** (*checksums*, véase la sección 9.4.3) indican errores. Estos bloques en mal estado no se utilizan.

A medida que un archivo aumenta de tamaño, el sistema operativo debe leer el bloque maestro de control para encontrar un bloque libre y luego actualizar el bloque maestro con la información correspondiente. Como inconveniente, este trabajo genera una gran cantidad de movimientos de las cabezas dado que el bloque maestro de control y los bloques libres rara vez (si es que alguna vez) se encuentran sobre la misma pista. Una solución utilizada en la práctica consiste en copiar el bloque maestro a memoria principal, realizar allí las actualizaciones y, periódicamente, actualizar el bloque maestro sobre el disco, en una operación de **sincronización** del mismo.

El hecho de tener dos copias del bloque maestro, una en memoria principal y otra en la unidad de disco, puede ocasionar la destrucción de la integridad de un disco si se apaga el sistema de computación antes de que se haya realizado la sincronización mediante la transferencia de la versión almacenada en memoria principal a la unidad de disco. El procedimiento convencional de apagado de computadoras personales y de otros sistemas,

incluye la sincronización mencionada. Por esta razón es tan importante el apagado de una computadora mediante el procedimiento normal en lugar del simple corte de la llave de alimentación eléctrica. De todos modos, aunque no se produzca una sincronización adecuada del disco, normalmente se dispone de una cantidad de información suficiente como para que un programa de recuperación de disco restituya la integridad del mismo, muy a menudo con la colaboración del usuario. (*Nota*: El Problema 8.12, al final del capítulo, propone una organización alternativa del bloque maestro de control que hace más sencilla la recuperación.)

8.5.2 Cintas magnéticas

Una unidad de cinta magnética tiene, habitualmente, una única cabeza de lectura-escritura, pero también puede tener cabezas para efectuar la lectura y escritura por separado. En la cinta magnética, un carrete de cinta plástica (*Mylar*) cubierta por un medio magnético pasa por debajo de la cabeza, que magnetiza la cinta en la escritura o detecta la información almacenada durante el proceso de lectura. La cinta magnética constituye un medio económico de almacenamiento de grandes cantidades de información, pero el acceso a cualquier sector en particular es lento debido a que se necesita que todas las secciones anteriores de la cinta pasen por debajo de la cabeza antes de encontrar la información requerida.

El almacenamiento de información en la cinta se realiza en un formato bidimensional, como el que se muestra en la figura 8.20. Los bits se almacenan en **palabras** ubicadas en forma transversal a la cinta, las que en el sentido longitudinal de la cinta constituyen **registros**. Un archivo está constituido por un conjunto de registros (habitualmente consecutivos). El registro es la menor porción de información que puede transferirse desde o hacia la cinta. Esto obedece a una razón más física que lógica. En el funcionamiento de una cinta, el sistema está normalmente en reposo. En el momento en que se desea escribir un registro en la cinta, un motor inicia el movimiento del sistema, lo que requiere un intervalo finito de tiempo. Una vez que la cinta llega a su velocidad, se escribe el registro y se detiene su movimiento, lo que nuevamente requiere un determinado tiempo. Los tiempos de arranque y detención del mecanismo consumen secciones de cinta, los que se conocen como **intervalos entre registros** (*interrecord gaps*).

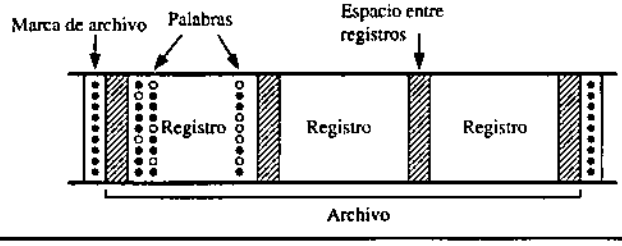


Figura 8.20 • Una porción de una cinta magnética.

Una cinta magnética es el elemento adecuado para el almacenamiento de grandes cantidades de datos, como copias de resguardo de discos o imágenes barridas (*scanned*), pero no sirve para la lectura y escritura de acceso aleatorio. Dos son las razones que justifican esta afirmación: una, el acceso secuencial puede requerir una gran cantidad de tiempo si la cabeza no se encuentra posicionada cerca del elemento que se está buscando. La segunda razón surge cuando se sobrescriben registros en el medio de la cinta, lo que normalmente no es una operación admisible en sistemas de cinta magnética. Si bien los registros individuales tienen todos el mismo tamaño, puede ocurrir que los intervalos entre registros se encimen con los registros debido a que los procesos de arranque y detención no son precisos.

Un **registro físico** puede dividirse en un número entero de **registros lógicos**. Por ejemplo, un registro físico que tiene un tamaño de 4.096 bytes puede estar compuesto por cuatro registros lógicos de 1.024 bytes cada uno. El acceso a los registros lógicos se maneja desde el sistema operativo, de modo tal que el usuario tenga la perspectiva de que el tamaño del registro lógico se relaciona directamente con el tamaño del registro físico, cuando lo que en realidad se transfiere a la cinta son los registros físicos. Por consiguiente, no existen intervalos entre registros lógicos.

Otra organización admite el uso de registros de longitud variable. En el comienzo de cada registro se coloca una marca especial que impide la confusión en cuanto al lugar en donde comienza cada uno de los registros.

8.5.3 Tambores magnéticos

Si bien en la actualidad son elementos casi totalmente obsoletos, tradicionalmente las unidades de tambor magnético han sido elementos más rápidos que los discos magnéticos. La ventaja en el rendimiento de los tambores se debe a que cuentan con una única cabeza fija en cada pista, lo que implica la inexistencia del tiempo correspondiente al movimiento de las cabezas cuando se determina el tiempo de acceso. La velocidad de rotación de un tambor puede ser mucho mayor que la del disco a causa de su forma cilíndrica angosta, la que, en consecuencia, también permite reducir la latencia de rotación.

La configuración de un tambor se muestra en la figura 8.21. La superficie externa del tambor se divide en una cantidad de pistas. Las caras inferior y superior no se utilizan para el almacenamiento, al igual que el interior del mismo, por lo que la capacidad de almacenamiento por unidad de volumen en un tambor es menor que en una unidad de disco.

El tiempo de transferencia de un sector de un tambor está determinado por el retardo de rotación y la longitud del sector. Dado que no hay movimiento de cabezas, no hay tiempo de búsqueda a considerar. En la actualidad, se suelen usar **discos de cabeza fija**, cuya configuración es similar a la de los tambores magnéticos, con una única cabeza por pista, pero mucho más económicos cuando se considera el costo de almacenamiento por byte, ya que se utilizan las superficies de los distintos platos que constituyen el disco en reemplazo de la cara externa del tambor.

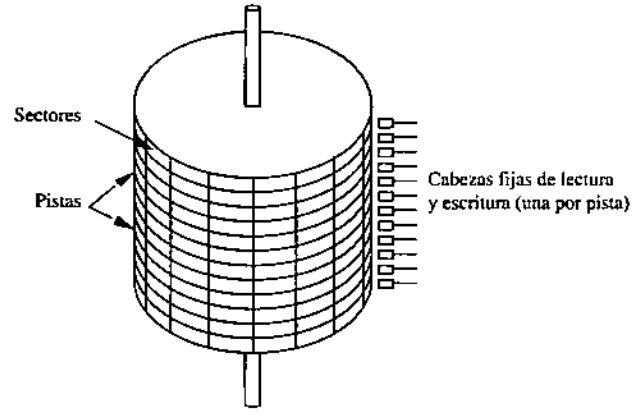


Figura 8.21 • Una unidad de tambor magnético.

8.5.4 Discos ópticos

Algunas tecnologías modernas aprovechan las virtudes de los mecanismos ópticos para el almacenamiento y recuperación de información. Tanto los **discos compactos** (CD, *compact disk*) como los más modernos **discos digitales versátiles** (DVD, *digital versatile disk*), que se analizan a continuación, emplean luz para leer información codificada en una superficie reflectiva.

El disco compacto

El disco compacto apareció en 1983 como soporte para la reproducción de música. Estos discos tienen la capacidad de almacenar 74 minutos de audio en formato estereofónico digital (dos canales). Se toman muestras del audio a razón de 2 x 44.000 muestras de 16 bits por segundo, lo que determina una capacidad del orden de 700 MB. Desde la introducción del sistema en 1983, la tecnología de discos compactos ha mejorado en lo que respecta a precios, densidades y confiabilidad, favoreciendo el desarrollo de **memorias de lectura en disco compacto** (CD-ROM, *CD read only memory*) para uso en computadoras, que ofrecen la misma capacidad de 700 MB. Su bajo costo (solo algunos centavos cada uno cuando se los produce en gran cantidad), sumado a su confiabilidad y alta capacidad han llevado a estos dispositivos a ser considerados una alternativa para la distribución de programas comerciales de computación, reemplazando a los discos flexibles.

Los discos compactos son básicamente solo de lectura debido a que se los estampa a partir de un disco maestro, en forma similar a la fabricación de los discos de audio. Un CD-ROM está constituido por un material plástico cubierto por aluminio, el que refleja en forma diferente la luz proveniente de **planos** (*lands*) y **huecos** (*pits*) que se corresponden, respectivamente, con zonas más altas y más bajas creadas en el proceso de estampado. El disco maestro se genera utilizando láser de alta potencia en un proceso de alta pre-

cisión. Los discos prensados (estampados) son menos confiables, por lo que se utiliza un esquema de corrección de errores complejo conocido como código de corrección de errores **Reed Solomon**. Los errores también se reducen por medio de la asignación de ceros y unos. Los unos se asignan a cada transición entre planos y huecos o entre huecos y planos, en tanto que los conjuntos de ceros se corresponden con las zonas planas. Esta asignación es distinta a la que se utiliza en la codificación Manchester, que asignaría ceros y unos a cada uno de las dos zonas.

A diferencia del disco magnético, en el cual todos los sectores ubicados en pistas concéntricas se alinean en forma similar a la de una torta cortada en porciones, y en donde la rotación del disco se realiza con **velocidad angular constante**, en los discos compactos la información se estructura en forma espiralada, como la de la figura 8.22, utilizando **velocidad lineal constante**. En esta espiral los huecos se ubican con el mismo espaciado desde un extremo del disco al otro. La velocidad de rotación, originalmente del orden de unas 30 RPM, se ajusta para que el disco se mueva más lentamente cuando la cabeza lectora está en los bordes que cuando está en el centro. Así, los discos compactos presentan el mismo problema que los discos flexibles en cuanto a sus elevados tiempos de acceso, debido a la gran latencia de rotación. Se pueden conseguir dispositivos de CD-ROM con velocidades de rotación de hasta 24X, siendo X la velocidad de rotación de un disco compacto de audio, lo que implica una disminución del tiempo de acceso promedio.*

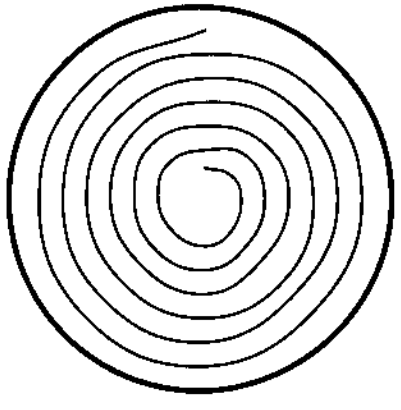


Figura 8.22 • Formato de almacenamiento en espiral en un disco compacto.

La tecnología de CD-ROM es apta para la distribución económica de grandes cantidades de información cuando se requieren muchas copias del mismo material, debido a que el costo de creación de un disco maestro y del prensado de las copias se distribuye entre la

* *N. de T.*: Los continuos avances tecnológicos hacen que el valor de 24X, común a la fecha de la edición original del presente texto (1999) haya quedado totalmente desactualizado frente a las velocidades de 52X y mayores disponibles a la fecha de esta traducción (octubre de 2001). Se presume, al igual que en el resto de la tecnología asociada, que la tendencia continuará en el mismo sentido.

totalidad de las copias producidas. Si solo se realizaran algunas pocas copias, el costo de cada disco sería alto porque los discos compactos no pueden prensarse en forma económica en bajas cantidades. Estos discos compactos tampoco pueden escribirse una vez que fueron prensados. (De todos modos, pueden fabricarse económicamente en bajas cantidades, utilizando grabadores de CD-ROM, de bajo costo, pero mediante un proceso diferente que produce discos de mucha menor vida útil.) Una tecnología nueva, utilizada para enfrentar este problema, es la del disco óptico WORM (*write once read many*) de **escritura única y múltiples lecturas**, en el que se utiliza un láser de baja densidad en el controlador de CD para escribir el disco óptico (una única vez en cada posición correspondiente a un bit). El proceso de escritura es más lento que el proceso de lectura, y el controlador y los medios son más caros que los correspondientes a los CD-ROM.

El disco digital versátil

Una versión más moderna del almacenamiento en discos ópticos está constituida por los **discos digitales versátiles** (DVD). Se han creado normas industriales para DVD para audio, para video y para almacenamiento de información DVD-ROM y DVD-RAM. Cuando se utiliza una única cara de un DVD, su capacidad de almacenamiento puede llegar a los 4,7 GB. Las mencionadas normas incluyen la posibilidad de almacenar información sobre ambas caras, en dos capas por cara, dando una capacidad total de almacenamiento de 17 GB. La tecnología de los DVD es un paso importante en la evolución a partir del CD, y no una tecnología totalmente nueva; de hecho, el reproductor de DVD es compatible hacia atrás, ya que además de reproducir DVD puede reproducir también discos compactos y CD-ROM.

Ejemplo: tiempo de transferencia de un disco rígido

Se pretende calcular el tiempo de transferencia de un disco magnético de un sistema de computación. En este ejemplo, se supondrá que el disco gira a razón de una vuelta cada 16 ms. El tiempo de búsqueda requerido para mover las cabezas entre pistas adyacentes es de 2 ms. Existen 32 sectores por pista, almacenados en orden lineal, no entrelazados, numerados desde el 0 al 31. La cabeza ve los sectores en ese orden.

Supóngase que la cabeza de lectura-escritura se encuentra ubicada sobre el comienzo del sector 1 de la pista 12. Existe un elemento de almacenamiento temporario cuyo tamaño es el suficiente como para almacenar una pista completa. Los datos se transfieren entre posiciones de disco mediante la lectura de la información original hacia la memoria, tras lo cual se ubica la cabeza en la posición de destino, donde se almacena la información.

Se desea calcular el tiempo que necesita la unidad para transferir íntegramente el sector 1 de la pista 12 a la pista 13.

Luego, se desea calcular el tiempo que se requiere para transferir todos los sectores de la pista 12 a los sectores correspondientes de la pista 13. Nótese que los sectores no necesariamente deben escribirse en el mismo orden en que han sido leídos.

Solución

El tiempo requerido para transferir un sector desde una pista a otra puede descomponerse en sus partes: el tiempo de lectura del sector, el tiempo de movimiento de la cabeza, el retardo de rotación y el tiempo de escritura del nuevo sector.

El tiempo requerido para leer o escribir un sector es simplemente el tiempo que tarda en pasar por debajo de la cabeza, el que se calcula como (16 ms/pista) x (1/32 pistas/sector) = 0,5 ms/sector. En este ejemplo, el tiempo de desplazamiento de la cabeza es de solo 2 ms debido a que la cabeza se mueve entre pistas adyacentes. Luego de leer el sector 1 de la pista 12, lo que lleva 0,5 ms, se requiere un retardo de rotación de 15,5 ms para permitir que la cabeza vuelva a alinearse con el sector 1. El tiempo de movimiento de las cabezas, de 2 ms, se superpone con los 15,5 ms de retardo de rotación, por lo que solo se debe tener en cuenta el mayor de ambos tiempos.

La suma de los tiempos individuales da por resultado que el tiempo requerido para transferir el sector 1 de la pista 12 al sector 1 de la pista 13 es de 0,5 ms + 15,5 ms + 0,5 ms = 16,5 ms.

El tiempo requerido para transferir toda la pista 12 a la pista 13 se calcula en forma similar. El *buffer* de memoria puede contener una pista completa, por lo que el tiempo necesario para la lectura o escritura de una pista íntegra coincide con el retardo de rotación de una pista, que es de 16 ms. El movimiento de la cabeza demora 2 ms, tiempo en el cual pasan cuatro sectores por debajo de la cabeza (a 0,5 ms por sector). Por consiguiente, luego de leer una pista y reposicionar la cabeza, la misma se encuentra sobre la pista 13, desplazada cuatro sectores del sector inicialmente leído en la pista 12.

Dado que los sectores pueden escribirse en un orden diferente al de su lectura, es posible iniciar la escritura de la pista 13 a partir del sector 5. El tiempo requerido para la escritura de la pista 13 es de 16 ms, por lo que el tiempo total requerido para la transferencia es de 16 ms + 2 ms + 16 ms = 34 ms. Nótese que se ha tomado como nulo el retardo de rotación debido a que cuando la cabeza se acomoda en la pista a ser escrita, aparece en el comienzo del primer sector a ser escrito. •

8.6 Dispositivos de entrada

Las unidades de disco, cinta y tambor son todos ejemplos de dispositivos de entrada y salida, que comparten la aplicación común de almacenamiento masivo. En esta sección se analizan algunos dispositivos que se utilizan exclusivamente para la entrada de información. Se inicia la descripción con uno de los dispositivos de mayor uso en computadoras: el **teclado**.

8.6.1 Teclados

El teclado de una computadora se utiliza para el ingreso manual de información. La figura 8.23 ilustra la distribución de teclas de un teclado acorde con la Norma ECMA-23 (2a. ed.). La distribución "QWERTY" (correspondiente a la asignación de las teclas de la fila superior izquierda D01-D06) concuerda con la distribución tradicional utilizada en máquinas de escribir. Las letras que se utilizan con mayor frecuencia se colocan lo suficientemente alejadas entre sí como para demorar a quien escribe y evitar problemas en las máquinas de escribir mecánicas. Si bien estos problemas no existen en los teclados electrónicos, se sigue utilizando la distribución tradicional.

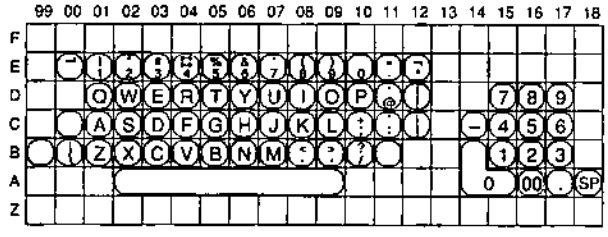


Figura 8.23 • Distribución de un teclado de acuerdo con la norma ECMA-23, 2a. ed. Las teclas para manejo de mayúsculas se colocan habitualmente en la fila B.

Cuando se digita un carácter, se genera un patrón binario que se trasmite a la computadora de destino. Si se codifican los caracteres en código ASCII de 7 bits, solo pueden utilizarse 128 combinaciones binarias. La mayor parte de los teclados que se expanden a partir de la norma básica ECMA-23 utilizan teclas modificadoras adicionales (*shift*, *escape* y *control*), por lo que no alcanza con un código de 7 bits. Existen diversas soluciones alternativas, entre las cuales ha ganado aceptación aquella que utiliza un patrón binario para cada una de las teclas modificadoras y otros patrones binarios para las teclas restantes.

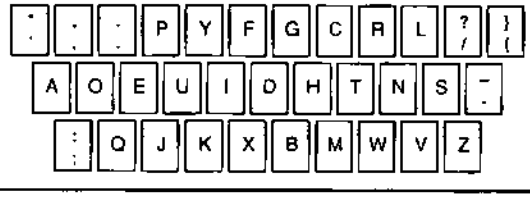


Figura 8.24 • La distribución del teclado Dvorak.

Otras modificaciones a los teclados basados en la norma ECMA-23 incluyen el agregado de teclas de función (como las de la fila F), y de teclas especiales, como las de tabulación (*tab*), eliminación (*delete*) y retorno al comienzo de la línea (*carriage return*). En otra modificación, que coloca las teclas más usadas en forma contigua, se obtiene el llamado **teclado Dvorak**, que se ilustra en la figura 8.24. A pesar de las ventajas de rendimiento del teclado Dvorak, no ha ganado aceptación.

8.6.2 Tabletas digitalizadoras

Una **tableta digitalizadora** es un dispositivo de entrada consistente en una superficie plana asociada con un **puntero** (*stylus*), similar a la que se ilustra en la figura 8.25. La tableta tiene incorporada una malla de cables que detectan la corriente inducida que se genera por causa del movimiento del puntero sobre la tableta. La misma transmite coordenadas X-Y (horizontal-vertical) y el estado de los botones del dispositivo, ya sea continuamente o solo en el caso de movimiento o pulsado de un botón, según sea el método de control utilizado. Las tabletas digitalizadoras se emplean frecuentemente para el ingreso de información desde mapas, fotografías, cartas o gráficos.

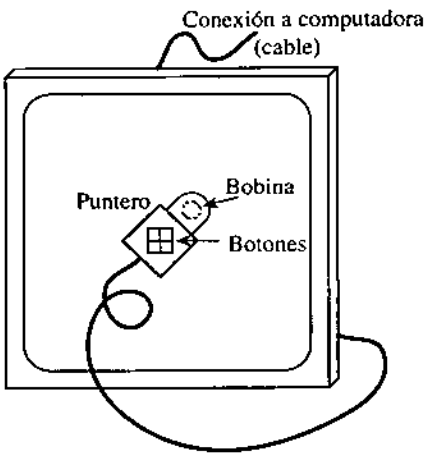


Figura 8.25 • Una tableta digitalizadora.

8.6.3 Ratones y trackballs

Un **ratón** (*mouse*) es un dispositivo de entrada manual que contiene una esfera de goma en la parte inferior y un conjunto de botones (uno o más) en la parte superior, como se ilustra en la figura 8.26 izquierda. A medida que se lo desplaza, la esfera ejecuta un movimiento de rotación proporcional al desplazamiento realizado. Los codificadores incluidos en el dispositivo detectan el sentido del movimiento y la distancia recorrida, las que se transmiten al procesador junto con el estado de los botones.

Un ***trackball*** puede verse como un ratón invertido. En este caso, el dispositivo en sí permanece estacionario mientras que la esfera se hace rotar en forma manual. La configuración del dispositivo se ilustra en la figura 8.26 derecha.

En el caso de los **ratones ópticos**, la esfera se reemplaza por un **diodo emisor de luz** (LED, *light emmiting diode*) y se utiliza una base especial que consiste en bandas horizontales y verticales reflectoras y absorbentes ubicadas en forma alternada. Se determina el movimiento por medio de la detección de las transiciones entre zonas reflec-

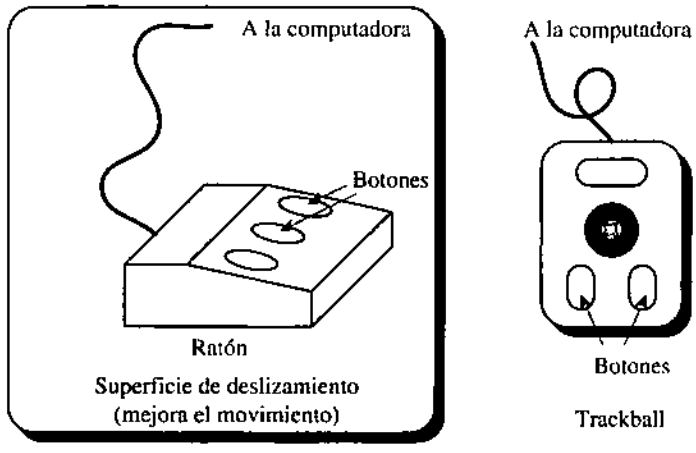


Figura 8.26 • Un ratón de tres botones (izquierda) y un *trackball* de tres botones (derecha).

toras y absorbentes. El ratón de tipo óptico no acumula suciedad tan fácilmente como el convencional, pudiendo ser usado en posición vertical y aun en ambientes no gravitatorios. No obstante, la rotación natural de la muñeca y del codo del usuario no coinciden con las bandas rectas verticales y horizontales del sistema óptico, por lo que el aprovechamiento efectivo del dispositivo requiere una cierta familiaridad por parte del usuario.

8.6.4 Lápices ópticos y pantallas sensibles al tacto

Los **lápices ópticos** y las **pantallas sensibles al tacto** (*touchscreens*) son dos dispositivos utilizados habitualmente para la selección de objetos. Un lápiz óptico no produce luz sino que detecta la luz proveniente de una pantalla de video, como lo muestra la figura 8.27. Un haz de electrones excita la capa de fósforo ubicada en la parte trasera de la superficie de la pantalla. El fósforo brilla y luego se apaga a medida que vuelve a su estado natural. Cada punto individual se refresca a una frecuencia de 30-60 Hz, de modo tal que el usuario percibe una imagen continua.

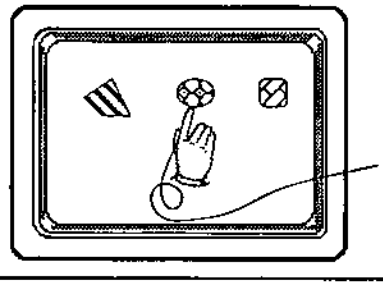


Figura 8.27 • Selección de un objeto con un lápiz óptico.

Cuando se refresca un punto opaco, el mismo se vuelve más brillante, con lo que el cambio de intensidad del brillo indica la posición del haz en un momento dado. Cuando el lápiz se encuentra ubicado en la posición en la que se produce el refresco del fósforo, la posición del haz de electrones determina la posición del lápiz. Dado que el lápiz detecta intensidad luminosa, solo puede distinguir entre áreas iluminadas. Las zonas oscuras de la pantalla aparecen todas iguales en tanto no hay cambio de su intensidad a lo largo del tiempo.

Las pantallas sensibles al tacto pueden adoptar dos técnicas, la fotónica y la eléctrica. En la figura 8.28 se ilustra una versión de la técnica fotónica. En la misma se cubre la pantalla con una matriz de rayos ubicados en forma vertical y horizontal. Si se interrumpe la continuidad de los rayos (por ejemplo, por la ubicación de un dedo), se puede determinar la posición a partir de los rayos que fueron interrumpidos.

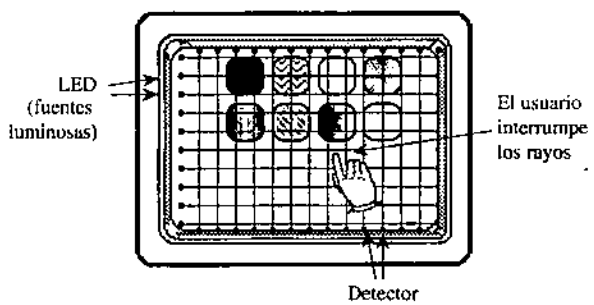


Figura 8.28 • Selección de un objeto sobre una pantalla sensible al tacto.

En una versión alternativa de la pantalla sensible, se cubre la pantalla con una superficie sensible al tacto. En este caso, el usuario debe hacer contacto con la pantalla si pretende registrar una selección.

8.6.5 Palancas de control (joysticks)

Una **palanca de control** (conocida popularmente por su nombre original inglés, ***joystick***) indica las posiciones horizontal y vertical por medio de la distancia en que se desplaza una varilla que sobresale de su base (véase la figura 8.29). Estos dispositivos se utilizan habitualmente en juegos de video y en la indicación de posiciones en sistemas gráficos. La palanca incluye potenciómetros ubicados en la base, los que convierten la posición X-Y en tensiones, las que a su vez se codifican en binario para ser ingresadas a un sistema digital. En aquellas palancas controladas por un resorte, la varilla vuelve a su posición central cuando se la libera. Si la varilla puede admitir el movimiento de rotación, se puede agregar al funcionamiento una dimensión adicional, como la altura.

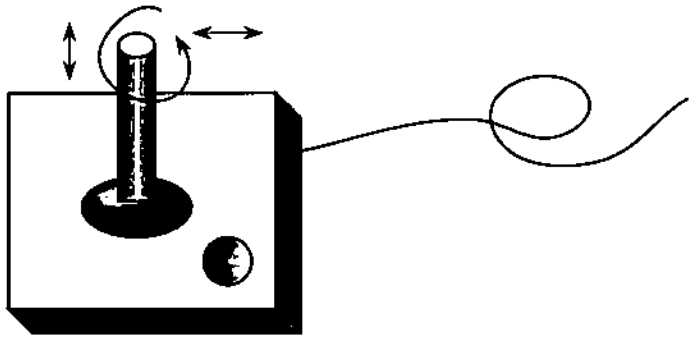


Figura 8.29 • Una palanca de mando con un botón de selección y una varilla rotativa.

8.7 Dispositivos de salida

Existen variados tipos de dispositivos de salida, de los cuales se analizan dos de los más comúnmente utilizados para la salida de información: las **impresoras láser** y las **pantallas de video**.

8.7.1 Impresoras láser

Una impresora láser está constituida por un tambor cargado sobre el cual actúa un rayo láser que descarga zonas seleccionadas de acuerdo con un patrón binario representativo de la página a ser impresa. A medida que el tambor avanza a lo largo de las líneas a relevar, las áreas cargadas recogen en forma electrostática el polvo entonador (tóner) sensible con que se realiza la impresión. El tambor continúa avanzando y el tóner se transfiere al papel, el que se calienta para fijar el polvo a la página. Una vez que la página está completa, el tambor se limpia de todo el material remanente y se vuelve a repetir el proceso para la página siguiente. La figura 8.30 muestra un diagrama esquemático del proceso descripto.

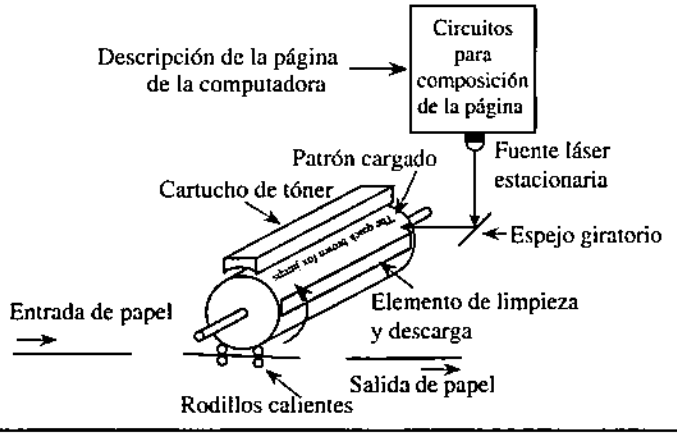


Figura 8.30 • Esquema de una impresora láser (adaptado de Tanenbaum, 2000).

Dado que el material (tóner) es una especie de plástico en lugar de tinta, no se absorbe en el papel sino que se funde sobre la superficie. Por esta razón, si se pliega una hoja de papel impreso en impresora láser, se producirán quiebres del material ubicado sobre el pliegue. Asimismo, suele ocurrir que el tóner se transfiera sin intención a otros materiales cuando se encuentra sometido al calor o a la presión (como en el caso de una pila de libros).

A diferencia de las viejas impresoras que solo permitían la impresión de caracteres codificados en ASCII, o, a lo sumo, de gráficos rudimentarios, las impresoras láser son capaces de imprimir todo tipo de información gráfica. Para permitir la transferencia de información desde la computadora hacia la impresora se han desarrollado diversos lenguajes. Uno de los más comunes es el lenguaje **Adobe PostScript**, basado en una pila, que es capaz de describir objetos tan diversos como caracteres ASCII, formas de alto nivel como círculos y rectángulos, y mapas de bits de bajo nivel. Se puede utilizar también para describir los colores de fondo y frente, y los colores requeridos para rellenar objetos.

8.7.2 Pantallas de video

Una pantalla de video, o **monitor**, consiste en un dispositivo de visualización luminiscente, como un tubo de rayos catódicos (CRT) o un panel de cristal líquido, y los circuitos de control del mismo. En la versión de rayos catódicos, dos placas de deflexión, horizontal y vertical, actúan sobre un haz de electrones que barre la pantalla del dispositivo en forma de **trama** (línea por línea, de izquierda a derecha, empezando desde arriba).

La figura 8.31 ilustra la configuración de un tubo de rayos catódicos. Por medio de un cañón electrónico se genera un flujo de electrones que impacta sobre una pantalla recubierta por fósforo, en posiciones controladas por las placas de deflexión horizontal y vertical. Los electrones tienen carga eléctrica negativa, por lo que si se aplica una tensión positiva en la grilla se produce la aceleración de los electrones hacia la pantalla, en tanto que una tensión negativa los aleja de la pantalla.

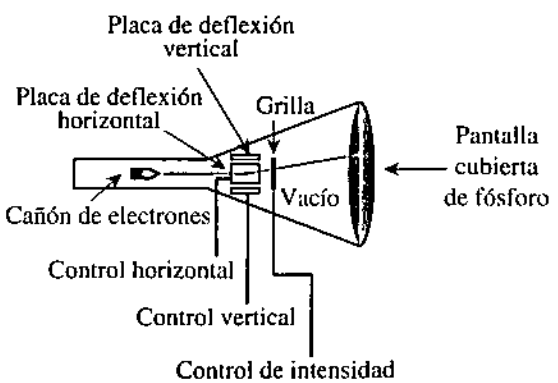


Figura 8.31 • Un tubo de rayos catódicos con un único cañón electrónico.

El color que se genera en la pantalla depende de las características del fósforo. En un tubo de rayos catódicos de color, suele haber tres tipos diferentes de fósforo (rojo, verde y azul) que se entrelazan en un patrón común, y tres cañones que producen tres haces que actúan en forma simultánea sobre la pantalla.

La figura 8.32 ilustra un controlador de pantalla sencillo. La escritura de la información sobre la pantalla se controla a través de un "reloj de puntos", el que genera un flujo continuo de ceros y unos alternados a una frecuencia que corresponde al tiempo en que un punto dado se encuentra activo sobre la pantalla. Cada uno de esos puntos es un **elemento de imagen** o **píxel**. El controlador de la figura 8.32 corresponde a una pantalla de 640 píxeles de ancho por 480 píxeles de alto. Se utiliza un contador de columnas, que en cada fila se incrementa desde 0 hasta 639 y se repite, y un contador de columnas que se incrementa repetitivamente desde 0 hasta 479. Las direcciones de fila y columna actúan como índices hacia la **memoria de video**, la que contiene los patrones binarios correspondientes a la imagen a ser presentada en la pantalla. El contenido de la memoria de video se transfiere a la pantalla a una frecuencia de entre 30 y 100 veces por segundo. Esta técnica de asignación de una zona de memoria RAM a la pantalla se conoce como **video asignado a memoria** (*memory mapped video*). Cada píxel individual de la pantalla puede estar representado por entre 1 y 12 bits del mapa de memoria. Cuando existe un solo bit por píxel, el mismo solo puede estar encendido o apagado, blanco o negro; múltiples bits por píxel permiten la asignación de distintos colores o tonalidades de gris a cada píxel.

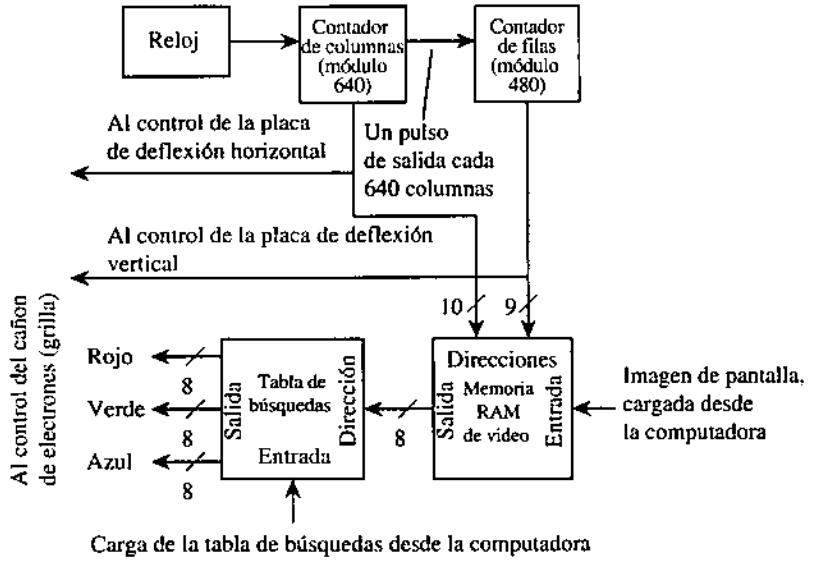


Figura 8.32 • Controlador de video para un monitor color de 640 x 480 píxeles de resolución (adaptado de Hamacher y otros).

El controlador de la figura 8.32 representa a cada píxel con ocho bits en la memoria, lo que indica que cada uno puede tener una de $2^8 = 256$ intensidades diferentes. En una configura-

ción simple se pueden repartir los ocho bits entre las entradas del tubo de rayos catódicos representativos del rojo, del verde y del azul (R, G y B), asignando tres bits al rojo, tres al verde y dos al azul. Una alternativa es la de utilizar una tabla de búsquedas en la cual la palabra de ocho bits se convierte en 256 diferentes palabras de 24 bits. De esos 24 bits de salida se asignan grupos de ocho bits para cada uno de los cañones correspondientes a los tres colores. Este método permite representar 2^{24} colores diferentes, pero dado que la tabla de búsquedas tiene solo 2^8 entradas, en cada momento solo pueden mostrarse 2^8 de las 2^{24} combinaciones (lo que se conoce como una paleta). La tabla de búsquedas se puede actualizar tantas veces como sea necesario para elegir distintos subconjuntos de los 2^{24} colores. Por ejemplo, para exhibir una imagen en tonos de grises, sin colores, se deben configurar R = G = B, por lo que se almacena para cada uno de los tres cañones una rampa con valores desde 0 hasta 255.

El ojo humano es relativamente lento si se lo compara con la velocidad de un dispositivo electrónico, y, por otra parte, no puede distinguir un quiebre en un movimiento que se produce a una frecuencia superior a los 25 Hz. Por lo tanto, la pantalla de una computadora solo requiere ser actualizada a una frecuencia de 25 o de 30 veces por segundo para lograr que el observador perciba una imagen continua. Si bien los monitores de video aplicados a las computadoras pueden tener cualquier frecuencia de barrido, definida por los diseñadores del monitor y de las placas de interfaz, cuando se los utiliza para aplicaciones televisivas se suele requerir una normalización de la velocidad de barrido. La televisión europea utiliza una frecuencia de 25 Hz, en tanto que en América del Norte es de 30 Hz.* Los tipos de fósforo usados en las pantallas no suelen tener largos tiempos de persistencia, por lo que las líneas de barrido se actualizan en forma entrelazada para evitar parpadeos. Así, las pantallas se actualizan a una frecuencia de 50 Hz en Europa y a 60 Hz en América del Norte, aunque solo se actualizan en cada barrido las líneas correspondientes a un campo en forma alternada. Para gráficos de alta resolución, puede actualizarse la pantalla completa a una frecuencia de 50 o 60 Hz, en lugar de plantear la actualización alternada. Existen quienes opinan que la elección europea de los 50 Hz no es buena, debido a que muchos observadores pueden distinguir los 50 Hz como un parpadeo en casos de baja iluminación o en los límites de la visión.

Por otra parte, los europeos plantean que la norma de transmisión **NTSC** utilizada por los Estados Unidos es inferior a la norma **PAL** europea, haciendo mención a que la sigla NTSC significa "nunca el mismo color, *never the same color*", debido a que presenta una menor capacidad para mantener la consistencia entre los colores de un cuadro y otro.

Las frecuencias de transmisión de información entre la computadora y el monitor de video pueden ser muy altas. Considérese que un monitor con una resolución de 1024 x 768 píxeles y 24 bits por píxel requiere, para una frecuencia de refresco de 60 Hz, **un ancho de banda** (cantidad de información que puede transferirse en un tiempo determina-

* *N. de T.*: En los países de América del Sur se encuentran ambas alternativas, en virtud de las diferentes normas utilizadas.

do) de 3 bytes/píxel x 1024 x 768 píxeles x 60 Hz, lo que da por resultado aproximadamente 140 Mbytes/s. Afortunadamente, la electrónica descripta asigna la memoria de video a la pantalla sin intervención de la CPU, aunque sigue siendo función de la CPU la generación de los píxeles hacia la memoria de video cuando se modifica la imagen de la pantalla.

Resumen

La entrada, la salida y las comunicaciones involucran la transferencia de información entre transmisores y receptores. En muchas oportunidades, los transmisores, los receptores y los métodos de comunicación no coinciden ni en la velocidad ni en el modo en que se representa la información, por lo que resulta una consideración importante la forma en que se deben compatibilizar los dispositivos de entrada, los de salida y el sistema que utiliza una comunicación dada.

Una estructura de bus ofrece un ancho de banda fijo que se comparte entre una cantidad de dispositivos. Se puede interconectar una estructura jerárquica de buses a través de puentes para permitir la ocurrencia simultánea de transferencias independientes. Desde la perspectiva del programador, las comunicaciones se pueden administrar por medio de un proceso de entrada-salida programada, o de entrada-salida administrada por interrupciones o por acceso directo a memoria.

Los dispositivos de almacenamiento masivo adoptan una variedad de formas. Son ejemplos de dispositivos de almacenamiento masivo los discos y las cintas magnéticas. Los dispositivos de almacenamiento óptico ofrecen mayor densidad de almacenamiento por unidad de superficie que los dispositivos magnéticos, pero son más caros y no tienen la misma capacidad de escritura por parte del usuario. El CD-ROM es un ejemplo de elemento de almacenamiento óptico.

Existe, además, una gran variedad de otros dispositivos de entrada-salida. Los pocos analizados en este capítulo que no son dispositivos de almacenamiento masivo pueden dividirse en dispositivos de entrada y de salida. Ejemplos de dispositivos de entrada son los teclados, las tabletas digitalizadoras, los ratones, los *trackballs*, las palancas de control, los lápices ópticos y las pantallas sensibles al tacto. Como ejemplo de dispositivos de salida se han presentado las impresoras láser y las pantallas de video.

Para lectura posterior

Pueden obtenerse hojas de datos y más literatura sobre dispositivos Intel, incluyendo los manuales de programador y de hardware de los procesadores Pentium, Pentium Pro y Pentium II, solicitándolos a Intel Literature Sales, P.O. Box 7641, Mt. Prospect, Il, 60056-7641, o bien, en los Estados Unidos y Canadá, llamando al (800) 548-4725. El material referido a la configuración Pentium II Xeon se obtiene en http://www.intel.com/design/chipsets/440zx.

V. C. Hamacher ofrece explicaciones sobre dispositivos de comunicaciones y periféricos, como un controlador alfanumérico para un tubo de rayos catódicos. A. Tanenbaum y W. Stallings plantean

también explicaciones claras acerca de dispositivos periféricos. El material referido a buses sincrónicos, asincrónicos y arbitraje está influido por la presentación de Tanenbaum.

Hamacher, V. C., Z. G. Vranesic y S. G. Zaky, *Computer Organization*, 3ª ed., Mc Graw-Hill, 1990.

Stallings, W., *Computer Organization and Architecture: Designing for Performance*, 4ª ed., Prentice Hall, 1996. (Traducción al español disponible: *Organización y arquitectura de computadores*, 5ª ed., Prentice Hall, 2000).

Tanenbaum, A., *Structured Computer Organization*, 4ª ed., Prentice Hall, 1999. (Traducción al español disponible: *Organización de computadoras, un enfoque estructurado*, 4ª ed., Prentice Hall, 2000.)

Problemas

8.1 ¿Cuál es el tiempo mínimo requerido para transferir 100 MB desde un dispositivo de audio a un procesador Pentium II, si se utilizan la arquitectura y los parámetros de la figura 8.7?

8.2 ¿Por qué debe asegurar la CPU que se deshabiliten las interrupciones antes de transferir el control a la rutina de atención de interrupciones?

8.3 Indicar la codificación Manchester para la secuencia binaria 10011101.

8.4 Un disco que contiene 16 sectores por pista usa un factor de entrelazado de 1:4. ¿Cuál es el menor número de revoluciones requerido para permitir la lectura de todos los sectores de una pista en secuencia?

8.5 Un disco magnético tiene dos caras. El área de almacenamiento de cada superficie tiene un radio interior de 1 cm y un radio exterior de 5 cm. Cada pista contiene la misma cantidad de bits, aun cuando cada pista difiere de todas las demás en su tamaño. La máxima densidad de almacenamiento del medio es de 10.000 bits/cm. El espaciado entre puntos homólogos de pistas adyacentes es de 0,1 mm, lo que incluye el espacio entre pistas. Supóngase que los espacios entre sectores son despreciables y que existe una pista en cada borde del área de almacenamiento.

a. ¿Cuál es la máxima cantidad de bits que pueden almacenarse en el disco?

b. ¿Cuál es la velocidad de transferencia de información desde el disco a la cabeza en bits por sector a una velocidad de rotación de 3.600 RPM?

8.6 Un disco tiene 128 pistas de 32 sectores cada uno, en cada cara de ocho platos. El disco gira a 3.600 RPM y requiere 15 ms para desplazar su cabeza entre pistas adyacentes. ¿Cuál es el máximo tiempo requerido para leer un sector cualquiera ubicado en cualquier lugar del disco?

8.7 Un disco de 300 Mbyte (300×2^{20} byte) de capacidad tiene 815 cilindros, con 19 cabezas, una velocidad entre pista y pista de 7,5 m/s, y una frecuencia de rotación de 3.600 RPM. El hecho de que haya 19 cabezas implica que existen 10 platos y que solo se usan 19 caras para el almacenamiento de datos. Cada sector contiene la misma cantidad de información y cada pista tiene la misma cantidad de sectores. El tiempo de transferencia entre el disco y la CPU es de 300 Kbytes/s. El espacio entre pista y pista es de 0,25 mm.

a. Calcular el tiempo necesario para leer una pista (no el tiempo requerido para transmitir el contenido de la pista a la computadora). Suponer que no se utiliza entrelazado.

b. ¿Cuál es el mínimo tiempo requerido para transferir la totalidad del contenido del disco a una CPU, bajo las mejores circunstancias posibles? Suponer que la cabeza de la primera cara a ser leída está posicionada al comienzo del primer sector de la primera pista y que se lee la totalidad de un cilindro antes de mover el brazo. Suponer, además, que la unidad de disco puede contener en memoria un cilindro completo, pero no más que eso. Durante la operación, la unidad de disco llena primero su memoria *buffer*, luego la descarga sobre la CPU y solo después de esa operación realiza una nueva lectura del disco.

8.8 Un disco de cabezas fijas tiene una cabeza por pista. Las cabezas no se mueven y, por ende, no hay componente correspondiente al movimiento de la cabeza al calcular el tiempo de acceso. Para este problema, calcular el tiempo requerido para copiar una cara a otra cara. Esta es una operación interna del disco y no requiere ninguna comunicación con el procesador central. Existen 1.000 cilindros, cada pista contiene 10 sectores y la velocidad de rotación del disco es de 3.000 RPM. Los sectores se encuentran todos alineados entre sí. Esto significa que dentro de un cilindro, el sector 0 de cada pista se encuentra alineado con el sector 0 de cada una de las restantes pistas, y que, dentro de una cara, el sector 0 se inicia sobre la misma línea radial dibujada desde el centro de la cara hasta su borde.

Una memoria *buffer* interna contiene un único sector. Cuando se lee un sector desde una pista, se lo almacena en la memoria *buffer* hasta que se lo escribe en otra pista. Solo entonces puede leerse otro sector. No es posible leer y escribir el *buffer* en forma simultánea, y el mismo debe cargarse o vaciarse en forma completa (no se admiten lecturas o escrituras parciales). Calcular el tiempo mínimo requerido para copiar una cara a otra, dadas las mejores condiciones de comienzo. Las caras deben ser imagen directa una de la otra. Esto es, el sector *i* de la cara de origen debe estar directamente encima o debajo del sector *i* de la cara de destino.

8.9 Calcular la capacidad de almacenamiento de una cinta magnética de 2.500 bytes/cm de densidad, de 20 metros de largo y 2.048 bytes en cada registro. El tamaño del espacio entre registros es de 1,25 cm.

8.10 En una pantalla asignada a memoria se tienen 1.024 píxeles de ancho por 1.024 píxeles de alto. La frecuencia de refresco es de 60 Hz, lo que significa que cada píxel se rescribe en la pantalla 60 veces por segundo, pero solo se rescribe un píxel por vez. ¿Cuál es el tiempo máximo disponible para la escritura de un único píxel?

8.11 ¿Cuántos bits se deben almacenar en la tabla de búsquedas de la figura 8.32? Si se elimina la tabla de búsquedas y se reemplaza la memoria RAM para entregar las salidas R, G y B de 24 bits en forma directa, ¿cuántos bits adicionales deben almacenarse en la memoria? Suponer que el tamaño inicial de la memoria es de $2^{10} \times 2^9 = 2^{19}$ palabras de 8 bits.

8.12 El bloque maestro de control que se presenta en la sección 8.2.1 mantiene el control de cada sector del disco. En una organización alternativa, que reduce en forma significativa el tamaño del bloque maestro de control, se almacenan los bloques en **cadenas**. La idea consiste en almacenar solo el primer bloque de un archivo en el bloque maestro de control y almacenar un puntero al bloque siguiente al final de cada bloque. Cada bloque subsiguiente se enlaza en forma similar.

a. ¿Cómo afecta esta solución al tiempo requerido para acceder al punto medio del archivo?

b. Luego de una caída del sistema, ¿podrá recuperarse más fácilmente un disco si solo se almacenara el primer sector del archivo en el bloque maestro de control, almacenando el listado de sectores restantes como encabezado al comienzo de cada archivo? ¿Cómo afecta esta solución al almacenamiento?

8.13 El lector es el administrador de un sistema de computación cuyo mantenimiento se realiza a través de Mega Equipment Corporation (MEC). Como parte de un mantenimiento de rutina, MEC realineó las cabezas de uno de los discos, y ahora el disco no puede leerse ni escribirse sin que produzca errores. ¿Qué pasó? ¿Podría darse esta situación con o sin el uso de una pista de sincronismo?

Capítulo 9

Comunicaciones

Se define como comunicación al proceso por el cual se transfiere información entre un origen y un destino. Los sistemas de comunicaciones cubren las distancias entre computadoras y pueden incluir a los sistemas de telefonía pública, de radio y de televisión. Los sistemas de comunicaciones de área extendida se han vuelto muy complejos, e incluyen todo tipo de combinaciones de voz, imagen y datos, los que se transfieren a través de cables, fibra óptica, radiofrecuencia y microondas. Las rutas de comunicación pueden atravesar las distancias por encima de la tierra, por debajo del mar, a través de celdas de radio locales y por vía satelital. La información que se origina como señales analógicas de voz puede convertirse en un flujo de datos digitales para así ser llevados en forma eficiente a través de grandes distancias, tras lo cual una nueva conversión recupera la señal analógica sin que aquellos que se comunicaron se enteren de la situación.

En este capítulo se tratarán las comunicaciones entre elementos ubicados a distancias dentro del rango del kilómetro, formando parte de **redes de área local** (LAN, *local area networks*), y también a través de distancias mucho mayores, para considerar las **redes de área extendida** (WAN, *wide area network*), tipificadas por Internet. El capítulo comienza analizando algunos conceptos referidos a comunicaciones.

9.1 Módems

La comunicación a través de las líneas telefónicas implica la generación de sonidos audibles, los que se convierten en señales eléctricas. Estas se transmiten a un receptor, en el que se convierten nuevamente en sonidos audibles. Esto no significa que la comunicación sobre una línea telefónica siempre requiera de alguien que hable y de alguien que escuche. Este medio audible de comunicación puede usarse también para transmitir información inaudible que se convierta a una forma audible.

La figura 9.1 ilustra una configuración en la que dos computadoras se comunican sobre una línea telefónica a través del uso de **módems** (el término surge como contracción de las palabras modulador y demodulador). Un módem convierte una señal eléctrica proveniente de una computadora en otra señal audible, con el propósito de su transmisión,

y luego realiza la operación inversa en la recepción. (Los módems no solo se usan para comunicaciones telefónicas, sino que también se los utiliza en otros sistemas de comunicaciones, como la transmisión de datos en redes de TV por cable).

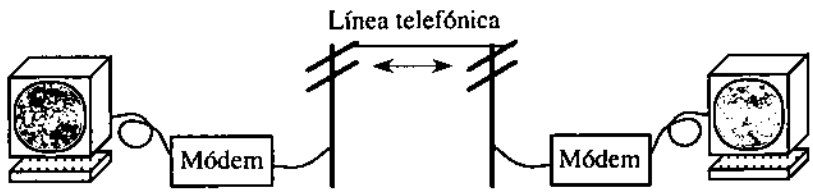


Figura 9.1 • Comunicación a través de una línea telefónica mediante módems.

Las comunicaciones por módem sobre una línea telefónica se realizan habitualmente en formato serie, transmitiendo un bit por vez. Los bits se codifican de manera adecuada al medio de transmisión. En las técnicas de comunicación existe una variedad de codificaciones de datos hacia el medio, que se conocen como esquemas de **modulación**. La figura 9.2 ilustra tres formas de modulación habituales.

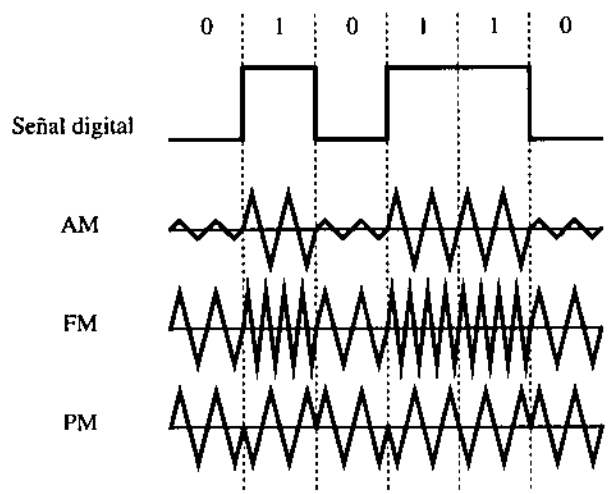


Figura 9.2 • Tres formas de modulación habituales.

La **modulación en amplitud** (AM) utiliza la intensidad de la señal para codificar los unos y los ceros. La modulación en amplitud lleva a implementaciones simples y de construcción económica. No obstante, dado que hay información en la amplitud de la señal, cualquier cosa que modifique la amplitud afecta a la señal. En el caso de un receptor radial que recibe una señal modulada en amplitud, existe una cantidad de situaciones que afectan la amplitud de la señal (como el pasar por debajo de un puente o cerca de líneas eléctricas, los relámpagos, etcétera).

La **modulación en frecuencia** (FM) no es tan sensible a los problemas asociados con la amplitud, debido a que la información se codifica en la frecuencia de la señal y no en

su amplitud. La señal modulada en frecuencia en una radio suele estar relativamente libre de estática y no se pierde cuando el receptor pasa por debajo de un puente.

La **modulación en fase** (PM) es la más utilizada en módems. En la misma, cuatro fases, separadas por 90° entre sí, duplican el ancho de banda de la información al transmitir dos bits por vez. El uso de la modulación en fase ofrece, además, cierto grado de libertad adicional al de la frecuencia, y resulta apropiado cuando se restringe la cantidad de frecuencias disponibles.

En la **modulación por impulsos codificados** (PCM) se toman muestras de la señal analógica y se la convierte a binario. La figura 9.3 ilustra el proceso de conversión de una señal analógica en una secuencia binaria PCM. La señal original se procesa al doble de la velocidad de la frecuencia más alta, lo cual produce valores a intervalos discretos. Las muestras se codifican en binario y se encadenan para producir la secuencia PCM.

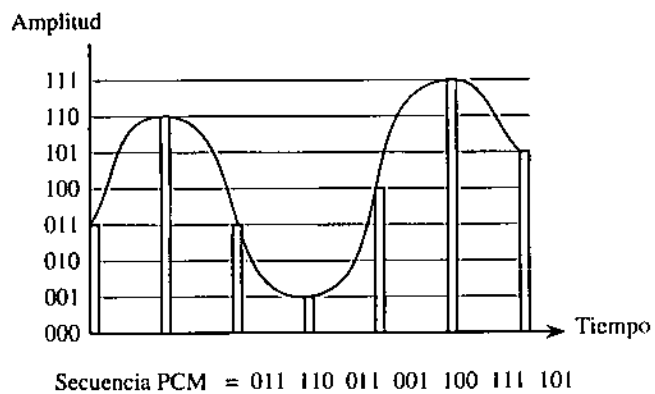


Figura 9.3 • Conversión de una señal analógica en una secuencia binaria modulada por impulsos codificados.

La modulación PCM es una solución digital, por lo que tiene todas las ventajas de los sistemas digitales de información. Si se requiere recuperar la señal, se lo puede hacer por medio del uso de repetidores ubicados a intervalos regulares. Si se disminuye la distancia entre los repetidores, se puede aumentar significativamente el ancho de banda efectivo de un canal. No obstante, las señales analógicas, a lo sumo, pueden ser adivinadas; por ende, solo pueden recuperarse en forma aproximada. No hay una buena manera de obtener señales analógicas perfectas en un ambiente ruidoso.

Se puede aplicar aquí la expresión de Shannon que define la frecuencia de datos en un canal ruidoso:

$$Frecuencia = ancho\ de\ banda \times log\ (1 + S/N)$$

Expresión en la que S representa la señal y N representa el ruido. Dado que las señales digitales pueden hacer uso de canales arbitrariamente ruidosos (en los que S/N es un valor grande) debido a su inmunidad al ruido, se pueden obtener tasas de transferencia más al-

tas sobre el mismo canal. Este es uno de los argumentos de peso para justificar el desplazamiento de la industria de las telecomunicaciones hacia la tecnología digital. La transición hacia los sistemas digitales también ha sido impulsada por la fuerte caída en los costos de los circuitos digitales.

9.2 Medios de transmisión

En un ambiente geográficamente cerrado, las computadoras pueden conectarse en red, en diferentes configuraciones, mediante cableados privados. Cuando los sistemas están geográficamente alejados, puede utilizarse la red publica de telefonía. Los usuarios se conectan a la red telefónica con módems que convierten niveles lógicos en sonidos audibles. El ser humano puede escuchar frecuencias que llegan hasta cerca de los 20 Khz, pero solo puede hablar en frecuencias que llegan hasta los 4 Khz, lo que coincide aproximadamente con el ancho de banda que la telefonía tradicional permite manejar en una línea con calidad de voz. Una señal analógica que deba aproximarse a una señal digital (voz, por ejemplo), debe muestrearse al menos dos veces por ciclo (para capturar sus valores alto y bajo); por lo tanto, la digitalización de una línea con calidad de voz requiere una frecuencia de muestreo de 8 Khz. A ocho bits por muestra, se obtiene una frecuencia de 8 bits/ciclo x 8 Khz = 64 Kbps (kilobits por segundo), que es lo que se puede encontrar en cualquier línea telefónica en América del Norte. Una muestra de cada ocho se utiliza para la administración de la línea, por consiguiente, la máxima frecuencia de transmisión posible sobre una línea con calidad de voz es de 56 Kbps.

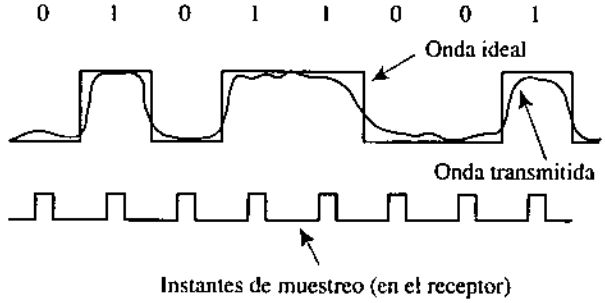


Figura 9.4 • Formas de onda ideal y transmitida.

La secuencia binaria transmitida se convierte en niveles alto y bajo, pero la forma de onda se atenúa y se distorsiona, tanto más a frecuencias altas y a grandes distancias. La figura 9.4 ilustra el problema del muestreo. El patrón binario 01011001 se ve representado por la forma de onda ideal, la que solo se aproxima por medio de la onda transmitida. La onda ideal tiene discontinuidades, difíciles de lograr en una onda real. En el análisis, se puede pensar en que la onda ideal se aproxima a través de la superposición de una serie de ondas sinusoidales, obteniéndose flancos más agudos en la medida en que se agregan frecuencias más altas.

Desafortunadamente, en la mayoría de los medios las frecuencias más altas se atenúan con mayor facilidad que las frecuencias más bajas. Si se suma esto al hecho de que las diferentes frecuencias se propagan a distintas velocidades, se tiene como resultado una distorsión de la señal a medida que se propaga. El grado de distorsión varía con el medio de transmisión, algunos de los cuales se describen a continuación.

9.2.1 Líneas bipolares abiertas

En uno de los esquemas más sencillos, un par de cables separados en el espacio transportan una señal y su referencia (la "tierra"). La configuración de una línea de dos cables abiertos se muestra en la figura 9.5a. Las líneas emiten radiación electromagnética y, además, recogen ruido, no necesariamente balanceado en la misma cantidad por cada línea, lo que distorsiona la señal diferencial. Las líneas también son vulnerables al acoplamiento capacitivo, o sea que recogen señales indeseadas de cables vecinos. La velocidad y la distancia requeridas para una transmisión confiable se limitan a alrededor de 19,2 Kbps y 50 m.

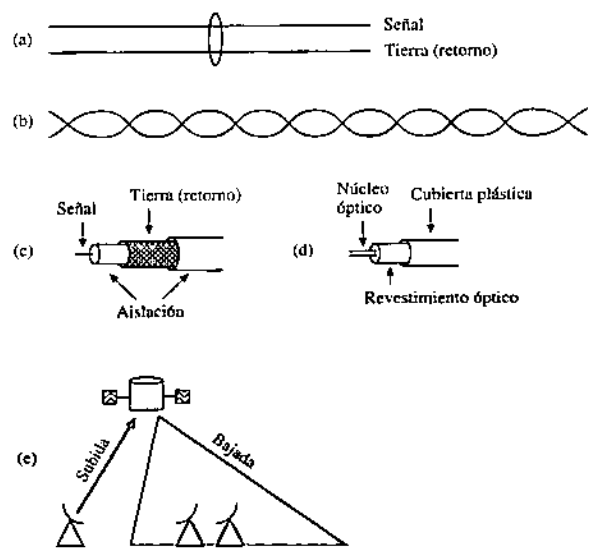


Figura 9.5 • Medios de transmisión: (a) línea bipolar abierta; (b) línea de par trenzado; (c) cable coaxial; (d) fibra óptica; (e) satélites.

9.2.2 Líneas de par trenzado

Si se trenzan las dos líneas de la configuración anterior, cualquier ruido externo espurio que se introduzca en la línea afecta a ambos cables, el de señal y el de referencia (tierra), en la misma proporción. La figura 9.5b muestra la configuración de líneas de par trenzado. La señal diferencial no se ve afectada y se pueden lograr transmisiones confiables hasta casi 100 m a 1 Mbps.

9.2.3 Cable coaxial

Para velocidades mayores (10 Mbps) y distancias mayores (cientos de metros), el cable de señal se coloca dentro del conductor de referencia (coaxialmente) con un aislador entre los dos, como se ve en la figura 9.5c. El tramado del conductor exterior hace que el cable sea más flexible. El objetivo es que el conductor interno esté efectivamente aislado de la interferencia externa, además de estar protegido contra pérdidas debidas a la radiación electromagnética.

9.2.4 Fibra óptica

La comunicación óptica es inmune a la interferencia electromagnética y a la modulación cruzada y, además, soporta un ancho de banda mucho mayor. Existe la necesidad de conversiones optoelectrónicas en cada extremo, lo que puede obtenerse en forma comercial hasta algunos Gbps (mediante diodos láser). Una fibra óptica está formada por su núcleo óptico, un revestimiento óptico y una envoltura plástica, tal como se ve en la figura 9.5d.

Un diodo emisor de luz (LED) es una fuente luminosa mucho más económica que un diodo láser, pero la luz que emite tiene ángulos variados. Por este motivo, se utilizan fibras **multimodo de índice discontinuo** (*multimode stepped index*) que reflejan la luz incidente por debajo del ángulo crítico nuevamente hacia el núcleo. Dado que las longitudes de los caminos difieren, el pulso recibido es más ancho, por lo que se logran nada más que velocidades de transmisión modestas. Los diodos emisores son baratos y sus tolerancias son un tema menos significativo que en el caso de los diodos láser.

Cuando se utiliza una fibra **multimodo con índice gradual**, la reflexión de la luz es mayor cuanto más se aleja del núcleo, lo que angosta el pulso y disminuye las pérdidas.

Las fibras de **modo único** reducen el diámetro del núcleo al equivalente de una longitud de onda, por lo que la luz viaja a través de un único camino libre de dispersión. Como fuentes de luz de este tipo de fibra se requieren diodos láser, lo que permite una operación a frecuencias de varios Gbps en distancias de decenas de kilómetros.

9.2.5 Satélites

Los satélites artificiales que se colocan en órbita alrededor del planeta Tierra se usan para comunicaciones cuando se requiere cubrir una zona amplia a un costo menor que el de una red cableada (incluyendo fibras ópticas). Los satélites naturales, dentro y fuera de la órbita terrestre (como la luna y los asteroides) podrían utilizarse para comunicaciones, pero en general no se los usa con dicho propósito.

En la comunicación satelital se transmite un haz colimado de microondas desde la tierra hacia un satélite, desde el cual se retransmite la señal al área cubierta en la Tierra por medio de un transceptor (*transponder*) que abarca una cierta banda de frecuencias. La configuración satelital se muestra en la figura 9.5e.

Un satélite típico tiene varios transceptores que funcionan a 500 MHz por canal. Que el área de cobertura sea pequeña significa que la señal transmitida tiene mayor intensidad y, por consiguiente, las antenas receptoras pueden tener menor tamaño. Esta es la configuración habitual de los sistemas satelitales de televisión de transmisión directa (DBS, *direct broadcast satellite*), en los que se utilizan antenas receptoras muy pequeñas. Los satélites del sistema DBS giran alrededor de la Tierra en una órbita muy baja, de aproximadamente 700 km de altura, por lo que se requiere un área de recolección menor que la necesaria para aquellos satélites que se colocan en orbita geoestacionaria (alrededor de 36.000 km por encima de la superficie terrestre), donde la fuerza gravitatoria de atracción de la Tierra y la fuerza centrífuga repelente se balancean, con lo que el satélite aparece como estacionario sobre la tierra cuando su órbita coincide con el Ecuador. Es por esto que las antenas satelitales de gran tamaño se apuntan en dirección al Ecuador.

Para las redes bidireccionales de comunicación vía satélite, se requiere que el retardo de la señal transmitida entre el usuario final y el satélite sea tolerable. La subida de la señal al satélite (*uplink*) es, normalmente, más lenta que la bajada (*downlink*). Esta característica se asemeja al modo típico de operación de un usuario final sobre una red como Internet, dado que menos del 10% del tráfico de la red va desde el usuario hacia la red, en tanto que el 90% del tráfico va desde Internet hacia el usuario. La velocidad de la comunicación está limitada por c (la velocidad de la luz en el vacío), cuyo valor es de 300.000 km/s. A lo largo de una distancia de 36.000 km, el retardo en el espacio abierto es de más de 100 ms hacia el satélite y otros 100 ms en su vuelta a la Tierra, además del retardo de procesamiento. Este valor es mayor que el retardo medio aceptable de 100 ms requerido en la respuesta a una acción sobre una tecla. Las órbitas de baja altura, del orden de solo 700 km, introducen retardos de propagación mucho menores, con lo que son más adecuadas para la implementación de redes interactivas.

9.2.6 Microondas terrestres

Los enlaces basados en la línea del horizonte terrestre son efectivos hasta distancias de alrededor de 50 km, particularmente cuando se requiere cruzar terreno dificultoso, aunque son sensibles a las perturbaciones atmosféricas, a las bandadas de aves, etcétera.

9.2.7 Radio

En un sistema de comunicación celular por radiofrecuencia, se coloca una estación radial de base en el centro de una **celda**, la que generalmente tiene un diámetro menor a 20 km. Dentro de la celda se utiliza una banda restringida de frecuencias para permitir la comunicación entre la estación de base y los dispositivos celulares móviles. Las celdas vecinas utilizan bandas distintas de frecuencia para que no haya confusión en la frontera entre ellas en el momento de una transferencia del usuario de una celda a la otra, lo que normalmente implica un cambio de frecuencias.

El ancho de banda total disponible en una celda es pequeño, del orden de 2 Mbps, y se subdivide en la cantidad de canales en uso. En zonas congestionadas, el tamaño de las celdas es menor que en zonas menos densamente pobladas y, en ciertos casos, no se extienden más allá del tamaño de un edificio.

9.3 Arquitectura de redes: redes de área local

Una **red de área local** (LAN) es un medio de comunicación que interconecta computadoras ubicadas geográficamente a una distancia de no más de algunos kilómetros. Una red de área local permite que un conjunto de computadoras y otros dispositivos, agrupados en un área reducida, compartan recursos comunes, como información, programas de aplicación, impresoras y dispositivos de almacenamiento masivo.

Una red de área local está constituida por circuitos (hardware), programas (software) y protocolos. El hardware puede adoptar la forma de cables y circuitos de interfaz. El software, normalmente, está incorporado en un sistema operativo, siendo el responsable de conectar al usuario con la red. Los protocolos son conjuntos de reglas que gobiernan los formatos, la temporización, la secuenciación y el control de errores. Los protocolos son elementos importantes para asegurar que la información se empaquete adecuadamente para su inyección en la red y que la misma se extraiga de la red en forma apropiada. La información a ser transmitida se descompone en partes, a cada una de las cuales se le agrega un **encabezado** que contiene información acerca de parámetros como el destino, el origen, los bits de protección contra errores y un sello horario. Esta información, a la que muchas veces se denomina **carga**, se combina con el encabezado para formar **un paquete** que se introduce en la red. Un sistema receptor cumple con el proceso opuesto, que permite extraer los datos del paquete.

El proceso de comunicación a través de una red, normalmente, se lleva a cabo en una cantidad de pasos estructurados en forma jerárquica, cada uno de los cuales tiene sus propios protocolos. Los distintos pasos deben seguir una secuencia cuando se transmite y la secuencia inversa cuando se recibe. Esto lleva a la noción de una **pila de protocolos**, en la cual el protocolo utilizado queda aislado dentro del escalón jerárquico correspondiente.

9.3.1 El modelo OSI

El modelo **OSI** (*Open System Interconnection*, Interconexión de Sistemas Abiertos) es un conjunto de protocolos establecido por la International Standards Organization (ISO) con la intención de definir y normalizar las comunicaciones de datos. El modelo OSI ha sido algo desplazado por el modelo TCP/IP vigente en Internet (véase la sección 9.5), pero sigue influyendo fuertemente sobre las comunicaciones en redes, en especial en la industria de las telecomunicaciones.

En el modelo OSI, el proceso de comunicación se divide en siete capas: las capas de **aplicación**, **presentación**, **sesión**, **transporte**, **red**, **enlace** y **física** (véase la figura 9.6).

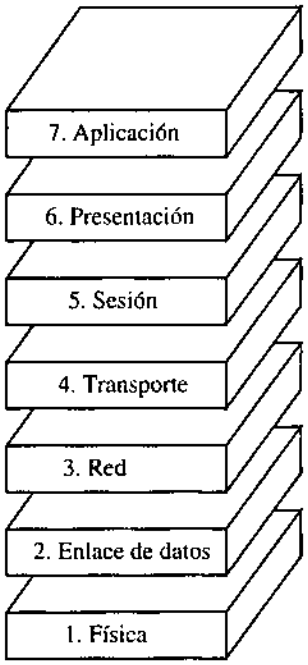


Figura 9.6 • Las siete capas del modelo OSI.

El modelo OSI no entrega una única definición acerca de la realidad del proceso de comunicación de datos. En cambio, el modelo OSI sirve como referencia para determinar cómo debería dividirse el proceso y qué protocolos deberían usarse en cada capa. El concepto permite que los proveedores de equipamiento seleccionen un protocolo en cada capa mientras aseguren la compatibilidad con el equipamiento de otros proveedores que puedan usar diferentes protocolos.

El nivel más alto del modelo OSI es la capa de aplicación; esta provee una interfaz que permite la comunicación de aplicaciones entre sí a través de la red. Ofrece soporte de alto nivel para aplicaciones que interactúan sobre la red, como los servicios de bases de datos para programas de bases de datos en red, el manejo de mensajes para programas de **correo electrónico** y el manejo de archivos para los programas de transferencia de archivos.

La capa de presentación asegura que las aplicaciones de comunicaciones vean todas un mismo formato de información. Esto se hace necesario debido a que distintos sistemas pueden usar distintos formatos internos de datos. Por ejemplo, algunos sistemas utilizan el formato *big-endian* para la representación numérica, en tanto que otros utilizan el formato *little-endian*.* La función de la capa de presentación es la de aislar a las aplicaciones de las diferencias que se producen en las capas inferiores.

* *N. de T.*: Estos dos términos se definieron en el capítulo 4, con respecto al orden en que se presenta la información que tiene más de un byte de tamaño.

La capa de sesión establece y termina las sesiones de comunicación entre procesos. Esta capa es responsable del mantenimiento de la integridad de las comunicaciones aun cuando las capas inferiores a ella pierdan datos. También sincroniza el intercambio y establece puntos de referencia para la continuación de una comunicación interrumpida.

La capa de transporte asegura transmisiones confiables entre origen y destino. Asigna los recursos de comunicaciones para que la información se transfiera en forma rápida y económica. La capa de sesión realiza pedidos a la capa de transporte, la que prioriza los pedidos y establece los compromisos entre velocidad, costo y capacidad. Por ejemplo, una transmisión puede ser dividida en varios paquetes, los que a su vez se transmiten sobre una cantidad de redes con el objeto de lograr una comunicación más rápida. En consecuencia, los paquetes pueden llegar a destino en forma desordenada, con lo que resulta responsabilidad de la capa de transporte el asegurar que la capa de sesión reciba la información en el mismo orden en que fue enviada. La capa de transporte provee las herramientas para la recuperación de errores entre origen y destino, así como el control del flujo (esto es, se asegura de que las velocidades de receptor y transmisor sean coincidentes).

La capa de red encamina la información a través de los sistemas intermedios y subredes. A diferencia de las capas superiores, la capa de red debe tener en cuenta la **topología** de la red, la que se define en función de la conectividad entre los componentes que la integran. La capa de red informa a la capa de transporte acerca del estado de las conexiones existentes y potenciales en la red, en términos de su velocidad, confiabilidad y disponibilidad. La capa de red está implementada con **encaminadores** (*routers*) que conectan distintas redes que usan el mismo protocolo de transporte.

La capa de enlace de información administra las conexiones directas entre los componentes de una red. Esta capa se divide en un **control lógico de enlace** (*logical link control*, LLC), el que es independiente de la topología de la red, y un **control de acceso al medio** (*media access control*, MAC), específico de la topología. En algunas redes, las conexiones físicas entre dispositivos no son permanentes, siendo responsabilidad de la capa de enlace el informar a la capa física cuándo debe realizar conexiones. Esta capa trabaja con **bloques** de datos (paquetes o conjuntos de paquetes que pueden estar entrelazados) que contienen direcciones, datos e información de control.

La capa física asegura que se realice la transmisión de la información cruda a través del medio físico, desde un elemento de origen hasta el elemento de destino. Transmite y repite las señales a través de los límites de la red. La capa física no incluye los circuitos propiamente dichos, pero sí incluye los métodos de acceso a los mismos.

9.3.2 Topologías

En las redes de área local existen tres topologías primarias, las que se ilustran en la figura 9.7. La topología en **bus** es la más simple de las tres. Los componentes de la red se conectan a un sistema tipo bus simplemente mediante su conexión al cable que corre a lo largo de la red o, en el caso de una red inalámbrica, por medio de la emisión de señales

sobre un medio común. Una ventaja de este tipo de topología es que permite la comunicación directa de cualquier componente con cualquier otro componente ubicado sobre el bus. Otra ventaja adicional es la relativa simpleza con la que se incorpora cualquier componente a la red. El control se distribuye entre los componentes y, por lo tanto, no hay un único componente de la red que sirva como intermediario, lo que reduce el costo inicial de este tipo de red. Como desventajas se puede plantear que incluye una limitación en la longitud del cable que va desde el bus hasta cualquier componente de la red (cuando se trata de una red alámbrica) y que a veces puede ser necesario cortar el cable para incorporar algún otro componente a la red, lo que interrumpe al resto. Ethernet es un ejemplo típico de red basada en una configuración de bus.

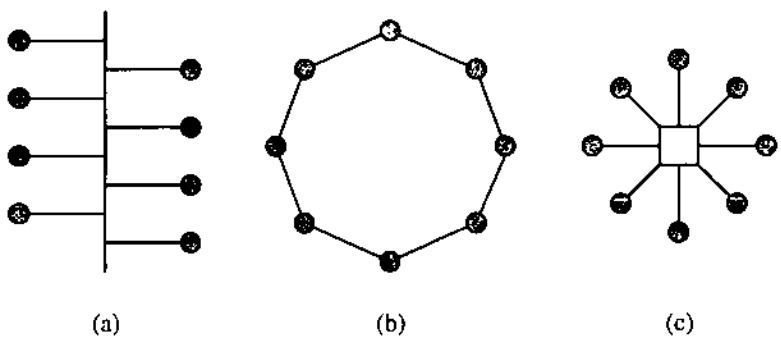


Figura 9.7 • Topologías de red en (a) bus; (b) anillo; (c) estrella.

La topología en **anillo** utiliza un único cable, cuyos extremos están unidos. Los paquetes se transfieren sobre la red a través de cada componente, hasta que alcanzan sus destinos. En cada destino, el paquete se extrae de la red, por lo que deja de circular por el anillo. Si durante su recorrido un paquete aparece nuevamente en su sistema de origen significa que la transmisión no resultó exitosa, en consecuencia, se detiene el envío y se realiza un nuevo intento. Un ejemplo de una red basada en una configuración de anillo es la red ***token ring*** de IBM.

En la topología en **estrella**, cada componente se conecta a un **nodo concentrador** (*hub*) central, el que sirve como intermediario de todas las comunicaciones que se producen sobre la red. En una configuración simple, el nodo recibe información desde un componente y la dirige hacia todos los demás componentes, dejando que cada uno de los componentes individuales determine si son o no el destino requerido. En una configuración más sofisticada, el nodo recibe la información y se la dirige a un componente específico de la red.

La ventaja de la configuración en estrella es que se concentra la mayor parte de los servicios, cambios de cableado y verificación de problemas (*troubleshooting*) en el nodo central. Una desventaja es que si ocurre una falla en el nodo se ve afectada la totalidad de la red. Otra desventaja es que, desde un punto de vista geométrico, la topología en estrella requiere mayor cantidad de cableado que las configuraciones en bus o en anillo, debido a que la conexión entre cada componente individual y el nodo se realiza a través de un

cable separado. Un ejemplo de topología en estrella se encuentra en la red ARCnet (a pesar de ser, en realidad, una red con configuración basada en bus).

9.3.3 Transmisión de datos

Las comunicaciones internas de un sistema de computación se sincronizan por medio de una señal de reloj común; por lo tanto, la transmisión de un cero o de un uno se indica a través de una señal de baja o alta tensión que se verifica en el momento indicado por la señal de reloj. Este esquema es simple pero no funciona correctamente en grandes distancias, como las que pueden corresponder a una red de área local. El problema es que no hay ninguna referencia temporal que permita indicar el comienzo o el final de un bit. Cuando se tiene una secuencia larga de ceros o de unos, la temporización puede desplazarse con respecto a las señales de reloj del transmisor y del receptor, debido a que estas señales de reloj no están sincronizadas en forma precisa. Las distancias en una red LAN son muy grandes como para mantener un reloj global en conjunto con un funcionamiento de alta velocidad. En consecuencia, las redes de área local utilizan generalmente el esquema de codificación Manchester (véase la sección 8.5) en el que se incorpora la señal de sincronismo dentro de la información.

La codificación Manchester se aplica en el nivel más bajo de la transmisión. En el nivel siguiente, el flujo de datos se descompone en paquetes y tramas que se transmiten a lo largo de la red, no necesariamente en orden. La capa de enlace es la responsable de descomponer el flujo de datos en paquetes, de armar los paquetes en tramas y de inyectar las tramas en la red. Cuando la capa de enlace recibe una trama, extrae de ella los paquetes y los configura en un formato que pueda ser utilizado por las capas de red de nivel superior. El tamaño de un paquete está habitualmente en el orden de un kilobyte, requiriéndose algunos microsegundos para la transmisión del mismo cuando se trata de distancias y velocidades habituales.

De las redes que utilizan configuración en bus, Ethernet es una de las predominantes. Para la transmisión, Ethernet utiliza un **acceso múltiple por detección de portadora** (*carrier sense multiple access*), con un esquema de **detección de colisiones** (CSMA/CD). Con esta configuración, cuando un componente de la red desea transmitir información, verifica primero el estado de la señal portadora. Si existe portadora en la línea, es que fue colocada por un dispositivo transmisor, por lo que el dispositivo que desea transmitir queda a la espera durante un tiempo aleatorio y vuelve a verificar la existencia de la portadora. Este período aleatorio de espera es importante porque permite evitar situaciones de **estancamiento** en las cuales todos los componentes que requieren acceder al bus quedan en permanente escucha y espera sincronizada.

Si no hay tráfico sobre la línea, la transmisión puede comenzar por medio de la colocación de una portadora sobre la línea, juntamente con la información a transmitir. La fuente transmisora también escucha para verificar la existencia de **colisiones**, dadas por dos o más componentes que transmiten en forma simultánea. Una colisión se detecta por

la presencia de más de una portadora. Estas colisiones pueden producirse debido al tiempo finito de propagación que le toma a cualquier señal recorrer la longitud del bus, que en una instalación Ethernet genérica puede ser de alrededor de 500 m. Cuando se produce una colisión, los componentes transmisores esperan durante intervalos aleatorios de tiempo antes de intentar una retransmisión.

La información transmitida viaja sobre el bus en ambas direcciones. Cada componente ve todos los paquetes de datos pero solo extrae aquellos paquetes cuyas direcciones de destino le corresponden. Luego de despachar exitosamente un paquete, el elemento de destino puede generar un acuse de recibo para el elemento que envió la información, generalmente en la capa de transporte. Si el despachante no recibe un reconocimiento luego de un período fijo de tiempo (que debe ser mayor que el tiempo requerido para un viaje completo por la red), transmite nuevamente el mensaje.

En la práctica, las colisiones deberían ser un hecho poco frecuente, con lo que la sobrecarga requerida para recuperarse de una colisión no es significativa. Normalmente, el rendimiento de una red Ethernet no se degrada seriamente hasta que el tráfico por la misma no aumenta a valores cercanos al 35% de la capacidad de la red.

9.3.4 Puentes (bridges), encaminadores (routers) y puentes de enlace (gateways)

A medida que las redes aumentan su tamaño, pueden dividirse en redes de menor tamaño interconectadas entre sí. Estas **subredes** de menor tamaño operan en forma casi independiente unas de otras y pueden usar distintas topologías y protocolos.

Si todas las subredes utilizan la misma topología y los mismos protocolos, puede darse el caso de que todo lo que se requiera para extender la red consista en elementos **repetidores**. Un repetidor amplifica las señales de la red, las que se van atenuando en relación con la distancia recorrida. La red principal se divide en subredes, en las que cada subred opera en forma relativamente independiente de las otras. Las subredes no son totalmente independientes debido a que cada una de ellas ve todo el tráfico que se produce en las demás. Una red con repetidores no puede extenderse a dimensiones muy grandes. Dado que el ruido se amplifica junto con la señal, si se utilizan demasiados repetidores encadenados, el nivel de ruido podría incluso dominar las señales.

Un **puente** hace más que amplificar niveles de señal. En un puente, los niveles individuales de las señales se restauran hacia niveles lógicos de 1 o 0, lo que impide la acumulación de ruidos. Los puentes tienen cierto nivel de inteligencia y, generalmente, pueden interpretar la dirección de destino de un paquete, para encaminarlo hacia la subred correspondiente. De esta manera, se puede reducir el tráfico sobre la red, dado que el método alternativo consistiría en enviar a ciegas cada paquete que se recibe a cada subred (como en el caso de una red basada en repetidores).

Si bien los puentes tienen cierto nivel de inteligencia que les permite analizar los bits que arriban y tomar decisiones de encaminamiento basadas en las direcciones de destino,

no pueden manejar protocolos. Un dispositivo **encaminador** (*router*) opera en un nivel superior, en la capa de red. Los dispositivos encaminadores conectan redes separadas lógicamente que utilizan el mismo protocolo de transporte.

Un **puente de enlace** (*gateway*) transfiere paquetes a través de la capa de aplicación del modelo OSI (capas 4 a 7). Su función es la de conectar redes disímiles a través de la conversión de protocolos y de formatos de mensajes, además de ejecutar otras funciones de alto nivel.

9.4 Errores de comunicación y códigos correctores de errores

En aquellas situaciones que involucran comunicaciones entre computadoras, y aun dentro de un sistema de computación, existe una posibilidad no nula de que se reciba información con errores, debido a ruidos en el canal de comunicaciones. Las representaciones de datos que se han considerado hasta el momento hacen uso de los símbolos binarios 1 y 0. En realidad, los símbolos binarios adoptan formas físicas, como tensiones y corrientes eléctricas. La forma física está sujeta al ruido que se introduce desde el ambiente, por ejemplo los fenómenos atmosféricos, rayos gamma y fluctuaciones de la alimentación, para nombrar solo algunas de las posibles causas. El ruido puede ocasionar errores, también conocidos como **fallas**, en las que un cero se convierte en un uno, o un uno se convierte en un cero.

Supóngase que un elemento transmisor envía a un receptor el carácter ASCII representativo de la letra 'b', y que durante la transmisión se produce un error que invierte el bit menos significativo de la palabra transmitida. La combinación binaria correspondiente a la 'b' es 1100010. La palabra binaria que llega al receptor es 1100011, que corresponde a la letra 'c'. No hay forma de que el receptor reconozca la existencia de un error a través de la simple observación del dato recibido. El problema en este caso consiste en que todas las posibles 2^7 combinaciones binarias del código ASCII representan caracteres válidos, y que si cualquiera de esas combinaciones se transforma en otra a causa de un error, la combinación binaria recibida aparenta ser válida.

Es posible hacer que el transmisor envíe bits adicionales de verificación junto con los bits de información. El receptor puede examinar estos bits de verificación y, ante ciertas condiciones, no solo puede detectar errores, sino también corregirlos. En las secciones siguientes se describen dos métodos para el cálculo de estos bits adicionales. Para comenzar, se introducen algunos datos y definiciones preliminares.

9.4.1 Tasa de error

Dentro de un sistema, los errores pueden aparecer de muchas maneras diferentes, además de que pueden adoptar muchas formas diferentes. Por el momento, se supondrá que la probabilidad de recibir con error un bit dado es independiente de la probabilidad de recibir con error algún otro bit cercano. En este caso, se puede definir la **tasa de error** como la

probabilidad de que algún bit determinado contenga un error. Obviamente, este número debería ser muy pequeño; en el caso de redes de fibra óptica, suele ser menor que 10^{-12} errores por bit. Esto significa, hablando ligeramente, que a medida que se analizan los bits recibidos, solo uno de cada 10^{12} bits puede ser erróneo. (En redes por radio, puede haber errores en uno de cada 100 paquetes.)

En el interior de un sistema típico de computación, la tasa de error puede estar por debajo de 10^{-18}. Si se plantea una estimación poco rigurosa, para una frecuencia de reloj de 500 MHz, y manejando 32 bits en cada ciclo de reloj, el número de errores por segundo en esa parte de la computadora será de 10^{-18} errores/bit x 500 x 10^6 palabras/s x 32 bits/palabra = 1,6 x 10^{-8} errores por segundo, lo que significa, aproximadamente, un bit erróneo cada dos años.

Por otra parte, si se recibe una secuencia de bits desde una línea de comunicación serie, a 1 millón de bits por segundo, y la tasa de error es en este caso 10^{-10}, la cantidad de errores por segundo será de 1 x 10^6 x 10^{-10} = 10^{-4} errores por segundo, lo que equivale a, aproximadamente, 10 errores por día.

9.4.2 Detección y corrección de errores

Uno de los métodos más simples y más antiguos para la detección de errores, utilizado ya en la transmisión y recepción de caracteres en los sistemas de telegrafía, consiste en agregar a cada carácter un **bit de paridad**, de valor 0 o 1, según fuera necesario para lograr que el número total de unos en la palabra fuese par o impar, según acordaran transmisor y receptor. En el ejemplo anterior, la transmisión de la letra 'b', codificada en ASCII y representada por la palabra 1100010, con un bit de paridad par, llevará un 1 como bit de paridad para que el total de unos resulte par. La palabra final a transmitir será 11000101.* El receptor, al examinar la palabra recibida, verificará la cantidad de unos que contiene; al encontrar una cantidad par de unos, podrá asumir que el carácter fue recibido sin errores. (Este método falla si existe una probabilidad importante de que se reciban dos o más bits erróneos en la palabra. En este caso, se requiere el uso de otros métodos, como los que se analizan más adelante.)

Códigos de Hamming

Con el agregado de bits adicionales a los caracteres de información, no solo es posible detectar errores, sino también corregirlos. Algunos de los códigos de corrección de errores más populares se basan en el trabajo realizado por Richard Hamming para Bell Telephone Laboratories (hoy Lucent Technologies).

* *N. de T.*: La palabra a transmitir varía según la posición que se asigne al bit de paridad. En el ejemplo, los autores suponen como bit de paridad el menos significativo. Si se utilizara el bit más significativo de la palabra, la combinación binaria a transmitir resultaría 11100010, que no altera la paridad aun cuando el bit de paridad se ubica en otra posición.

Se pueden detectar errores simples (un error en un único bit de una palabra) en el código ASCII por medio del agregado de un bit de redundancia en cada **palabra** del código. La **distancia de Hamming** define la distancia lógica entre dos palabras válidas, medida por la cantidad de dígitos que difieren entre esas palabras. Si en un carácter ASCII se modifica un solo bit, el patrón binario resultante representa un carácter ASCII distinto. La distancia de Hamming en este código es 1. Si se recodifica la tabla del código ASCII para que exista una distancia de 2 entre palabras válidas, entonces, para que un carácter válido se convierta en otro carácter válido, deben cambiar dos bits. Por consiguiente, se puede detectar un error simple porque la palabra afectada estará fuera del rango de caracteres válidos.

Una forma de codificar el código ASCII para convertirlo en un código de distancia 2 requiere la asignación de un bit de paridad, el que toma el valor 0 o 1 necesario para que el total de unos en la palabra sea siempre par o impar. Si se utiliza **paridad par**, el bit de paridad para la letra 'a' será un 1, dado que la palabra binaria que representa a la 'a' contiene tres unos: 1100001. La asignación de un bit de paridad par, de valor 1, y ubicado ahora a la izquierda de la palabra (en la posición más significativa), hará que la cantidad total de unos de la palabra sea par, dando como representación de la 'a' la palabra binaria 11100001. Esta representación se ilustra en la figura 9.8. En forma similar, el bit de paridad requerido para codificar la letra 'c' es 0, lo que lleva a la palabra 01100011. Si, por el contrario, se utiliza **paridad impar**, los bits de paridad adoptan los valores opuestos a los mencionados: 0 para la 'a' y 1 para la 'c', con lo que en este caso ambos caracteres quedan codificados, respectivamente, como 01100001 y 11100011.

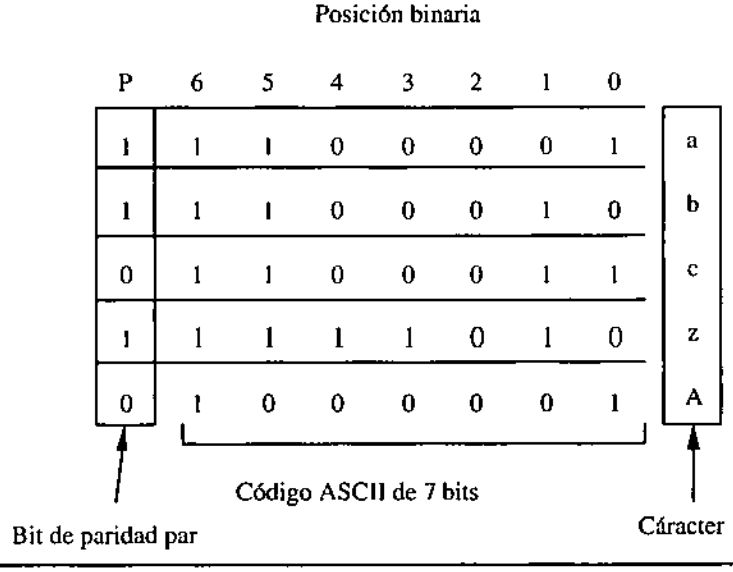


Figura 9.8 • Agregado de paridad par a algunos caracteres codificados en ASCII.

La tabla del código ASCII con paridad tiene ahora $2^8 = 256$ entradas, la mitad de las cuales (para paridad par, las que tienen una cantidad impar de unos) representan combinaciones inválidas para el código. Si recibe una combinación inválida, el receptor sabe que se ha producido un error, por lo que puede requerir una retransmisión.

El hecho de retransmitir la información no siempre resulta práctico, y, en estos casos, sería más cómodo poder detectar y corregir un error. El uso de un bit de paridad permite detectar un error pero no permite ubicar la posición del mismo. Si en un sistema que funciona con paridad par se recibe la combinación binaria 11100011, se detecta la existencia de un error debido a que la paridad es impar. No hay información suficiente para determinar si la palabra original era una 'a', una 'b', o algún otro carácter de la tabla del código. De hecho, el carácter original hasta podría haber sido una 'c' con el error ubicado en el bit de paridad.

Con el objeto de construir un código corrector de errores que sea capaz de detectar y corregir un error simple, se debe agregar un nivel de redundancia mayor que el que puede aportar al código ASCII original un único bit de paridad. Esto significa que la cantidad de bits con los que se codifica cada uno de los caracteres aumenta todavía más. Considérese, por ejemplo, la combinación binaria 1100001, representativa de la letra 'a'. Para detectar y corregir un único error en cualquier posición de la palabra, es necesario asignar siete patrones binarios a las palabras que se obtienen a partir de la 'a' original si se altera un bit: 0100001, 1000001, 1110001, 1101001, 1100101, 1100011 y 1100000. Se puede hacer lo mismo para la letra 'b' y para los demás caracteres del código, pero se debe construir el código de forma tal que no haya una palabra binaria común a más de un carácter ASCII original; si así no ocurriera, no habría forma de determinar sin ambigüedad la combinación binaria original.

El problema derivado de la utilización de redundancias en esta forma es que se asignan ocho palabras binarias para cada carácter a codificar: una para la palabra original y siete para los patrones adyacentes derivados de un error. Dado que el código ASCII está formado por 2^7 caracteres, y dado que hacen falta 2^3 palabras binarias por cada carácter, solo se pueden recodificar $2^7/2^3 = 2^4$ caracteres si se utilizan para la representación nada más que los siete bits originales.

Con el objeto de poder recodificar todos los caracteres, se deben agregar bits adicionales de **redundancia** a las palabras del código (también conocidos como bits de **verificación**). Para determinar cuántos bits se requieren, se supondrá que la palabra a recodificar tiene k bits. Se utilizan r bits de redundancia, debiendo cumplirse la siguiente relación algebraica:

$$2^k \bullet (k + r + 1) \leq 2^{k+r} \equiv k + r + 1 \leq 2^r$$

La expresión planteada surge del siguiente razonamiento: para cada una de las 2^k palabras originales, existen k combinaciones binarias que corresponden a la palabra original modificada en un bit, más r combinaciones binarias que corresponden a la posibilidad de

que los bits errados sean los de paridad, más la combinación binaria correspondiente a la palabra binaria original, sin error. Por consiguiente, el código corrector de errores tendrá un total de $2^k \times (k + r + 1)$ combinaciones binarias diferentes. Con el objeto de poder admitir todas estas combinaciones, el código requiere que exista una cantidad suficiente de combinaciones producidas por $k + r$ bits. En consecuencia, la cantidad de combinaciones binarias del código corrector debe ser a lo sumo igual a 2^{k+r}. El código ASCII está constituido por palabras de 7 bits, por lo que se requiere resolver la expresión anterior para r, considerando $k = 7$. Haciendo pruebas con valores sucesivos a partir de $r = 1$, se encuentra que el menor valor de r que satisface la expresión mencionada es $r = 4$. En consecuencia, el código resultante tendrá palabras de 11 bits.

Bits de verificación C8 C4 C2 C1	Posición binaria verificada
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7
1 0 0 0	8
1 0 0 1	9
1 0 1 0	10
1 0 1 1	11

Figura 9.9 • Bits de verificación en un código que corrige un solo error para el código ASCII.

A continuación, se debe determinar la forma de convertir la tabla del código ASCII en un código de 11 bits. El objetivo es el de asignar los bits de redundancia a las palabras originales en forma tal que permita la identificación de cualquier error simple. Una forma de hacer la asignación se muestra en la figura 9.9. A cada uno de los 11 bits de la palabra modificada se le asigna una posición en una tabla indexada por su posición, de 1 a 11. Asociado a cada índice se muestra la representación binaria en cuatro bits de los números enteros del 1 al 11. Con esta asignación se puede observar que, al leer cada una de las once filas de los cuatro bits de verificación, cada fila representa una combinación única de unos y ceros; dicho de otra forma, no hay dos filas iguales. Por ejemplo, la primera fila tiene un único 1 en la posición C1, situación que no se da en ninguna otra fila (existen filas que tienen un 1 en la posición C1, pero también tienen unos en alguna otra posición de los bits de verificación).

Si ahora se lee en forma vertical cada una de las cuatro columnas de verificación, las posiciones de los unos indican cuáles son los bits de la palabra, que se encuentran detallados en la columna de la derecha de la tabla, que deben incluirse en un mismo grupo de bits que deberá tener paridad par. Por ejemplo, el bit de paridad C8 cubre un grupo de 4 bits ubicados en las posiciones 8, 9, 10 y 11, los que, en su conjunto, deben tener pari-

dad par. Si se cumple esta característica cuando se transmite la palabra de 11 bits, pero se produce un error en la transmisión y el conjunto llega al receptor con paridad impar, el receptor sabrá que el error debe haberse producido en alguna de las posiciones 8, 9, 10 u 11. Como se verá, la posición exacta se determina a partir del análisis de los restantes bits de verificación.

En mayor detalle, cada bit de la palabra de 11 bits, modificada mediante la inclusión de los bits de verificación, se asigna a una combinación única de los cuatro bits de verificación C1, C2, C4 y C8. Las combinaciones se calculan como la representación binaria de la posición del bit que se está verificando, a partir de la posición 1. Por consiguiente, C1 se encuentra en la posición 1, C2 en la posición 2, C4 en la posición 4, etc. Los bits de verificación pueden colocarse en cualquier posición de la palabra, pero, normalmente, se los ubica en las posiciones correspondientes a las potencias de 2 para simplificar el proceso de localización del error. Este código particular se conoce como **código de corrección de un único error** (SEC, *single error correcting code*).

Dado que las posiciones de los unos es única en cada una de las combinaciones de los bits de verificación, se puede localizar el error mediante la simple observación de cuáles son los bits de verificación que dan error. Considérese el formato de la figura 9.10, en el que se representa la letra 'a'. Los valores de los bits de verificación se determinan en la forma indicada en la tabla de la figura 9.9. El bit de verificación C1 = 0 determina paridad par sobre el grupo de bits {1, 3, 5, 7, 9, 11}. Los elementos del grupo se obtienen a partir de las posiciones que tienen unos en la columna C1 de la figura 9.9. En forma similar, el bit de verificación C2 = 1 determina paridad par en el grupo de bits {2, 3, 6, 7, 10, 11}. Asimismo, el bit de verificación C4 = 0 genera paridad par sobre el grupo de bits {4, 5, 6, 7}. Finalmente, el bit de verificación C8 = 0 establece paridad par para el grupo de bits {8, 9, 10, 11}.

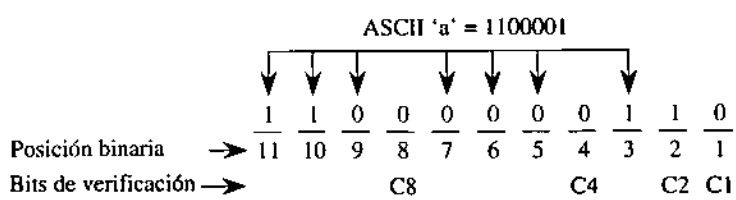


Figura 9.10 • Formato de un código ASCII con corrección de un único error.

Como alternativa para la determinación de los integrantes de un grupo de paridad en una tabla, el bit n de la palabra codificada se verifica por medio de aquellos bits de verificación que se encuentran en las posiciones $b_1, b_2, \ldots b_j$, tales que $b_1 + b_2 \ldots + b_j = n$. Por ejemplo, el bit 7 se verifica con los bits de las posiciones 1, 2 y 4, dado que 1 + 2 + 4 = 7.

Supóngase ahora que el receptor recibe la combinación binaria 10010111001. Si se utiliza el código corrector de errores recién descripto, ¿cuál fue el carácter enviado? Se comienza calculando la paridad de cada uno de los bits de verificación de acuerdo con lo que se muestra en la figura 9.11. Como se puede observar, los bits C1 y C4 tienen pari-

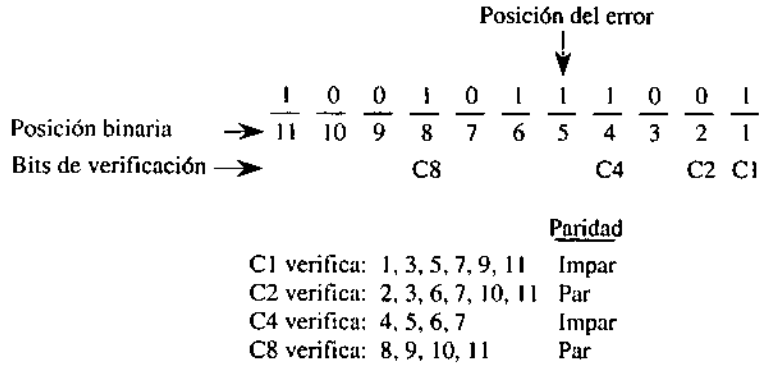


Figura 9.11 • Cálculo de paridad para un carácter ASCII en un código corrector de un único error.

dad impar. Con el objeto de localizar el error, simplemente se requiere sumar las posiciones de los bits que hayan dado paridad impar. En consecuencia, el error está en la posición 4 + 1 = 5. La palabra que fuera enviada es 10010101001. Si se eliminan los bits de verificación, se obtiene la combinación binaria 1000100, lo que corresponde al carácter ASCII 'D'.

Una forma de interpretar los códigos correctores de un error es pensando que las combinaciones válidas del código están separadas entre sí lo suficiente como para que un único error ubique a la palabra no perteneciente al código más cerca de una combinación válida del código que de las demás. Considérese, por ejemplo, un código para corregir un único error formado solo por dos símbolos: {000, 111}. Las relaciones de distancia de Hamming entre todas las posibles combinaciones de tres bits se muestran en la figura 9.12. El cubo tiene dimensiones mayores cuando las palabras tienen mayor tamaño, dando por resultado lo que se denomina un **hipercubo**. Las dos palabras válidas se encuentran en vértices opuestos. Cada error simple genera una palabra inválida ubicada en un vértice diferente del cubo. Cada palabra con error tiene una única palabra válida cercana, lo que permite la corrección de un único error.*

Códigos detectores de dos errores y correctores de un único error

Si se considera un caso en el que puedan producirse dos errores, se podrá ver que el código corrector de un error simple analizado funciona como **detector de dos errores** (DED, *double error detection*) pero no como **corrector de dos errores** (DEC, *double error correction*). Dado que la distancia entre palabras válidas del código es 3, dos errores en una palabra válida se detectarán como una palabra inválida en el cubo, por lo que

* *N. de T.*: Partiendo del concepto de distancia establecido al principio de esta sección, el procedimiento descripto genera un código de distancia 3 (dos combinaciones válidas del código difieren, como mínimo, en tres bits). Un único error deja la palabra que lo contiene a distancia 1 de una única palabra del código, y a distancia 2 de las restantes, por lo que la determinación del error es inmediata.

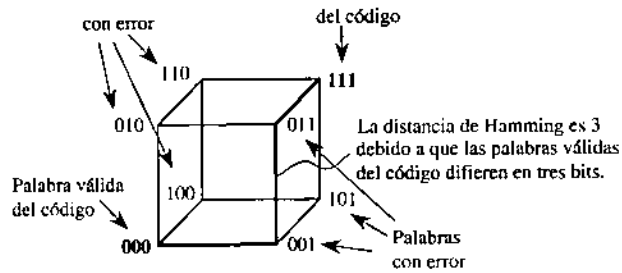


Figura 9.12 • Relaciones de distancia de Hamming entre palabras de un código de tres bits. Las palabras válidas son 000 y 111. Las combinaciones restantes representan errores.

se puede lograr la detección de dos errores simultáneos. No obstante, es imposible determinar en forma indudable la palabra original, debido a que los vértices que corresponden a una palabra con dos errores se superponen con los vértices correspondientes a otra palabra con un solo error. Por consiguiente, todo código corrector de un solo error es también un código detector de dos errores, aun cuando no todo código detector de dos errores sea un código corrector de un único error. Con el propósito de corregir dos errores se requiere una distancia de Hamming de 5. En general, un código que pretenda detectar p errores requiere una distancia igual a $p + 1$, en tanto que si se desean corregir p errores, la distancia requerida es de $2p + 1$.

9.4.3 Control de redundancia vertical

El código corrector de errores simples descripto en la sección anterior se usa para la detección y la corrección de errores simples en palabras de datos individuales. Se agregan bits de redundancia a cada palabra de datos y cada palabra resultante se trata en forma independiente. El esquema de recodificación suele conocerse como un **control de redundancia horizontal** o **longitudinal** (LRC, *longitudinal redundancy check*) debido a que la palabra del código se ve extendida a lo ancho con el agregado de los bits de redundancia.

Una solución alternativa considera el uso de un código de **control de redundancia vertical** (VRC, *vertical redundancy checking*). Este planteo involucra el agregado de una palabra de **verificación de paridad** (*checksum*) al final de cada grupo de palabras a transmitir. En este caso se calcula la paridad en base a cada una de las columnas, con lo que se obtiene una palabra de paridad que se agrega al mensaje. La palabra de paridad debe ser calculada y agregada por el transmisor, tras lo cual será recalculada por el receptor, el que deberá compararla con la palabra de paridad transmitida originalmente. Si se detecta un error, el receptor deberá requerir una retransmisión dado que no existe la suficiente redundancia como para identificar la posición del error. Con el objeto de mejorar la verificación de errores pueden combinarse los cálculos de paridad longitudinal y vertical, como se puede ver en la figura 9.13, en la que se calculan ambas paridades para un bloque de datos que incluye las letras que van desde la 'A' hasta la 'H'.

P	*Código*	*Carácter*
0	1000001	A
0	1000010	B
1	1000011	C
0	1000100	D
1	1000101	E
1	1000110	F
0	1000111	G
0	1001000	H
1	0001000	Verificación

Figura 9.13 • Detección combinada por medio de controles de redundancia vertical y longitudinal. Los bits de verificación forman paridad par en cada columna.

En algunos casos, los errores se producen por bloques, con lo que pueden afectar a varios bits adyacentes tanto en forma horizontal como vertical. Para esta situación resulta más apropiado el uso de un esquema más poderoso, como el del **control de redundancia cíclica** (CRC, *cyclic redundancy check*), que es una variante del control vertical, en donde los bits se agrupan en una forma particular, la que se describe en la próxima sección.

9.4.4 Control de redundancia cíclica

El control de redundancia cíclica (CRC) es un esquema de detección y corrección de errores más poderoso que los anteriores, que puede operar en presencia de **bloques de errores** –conjuntos de bits que comienzan y terminan en un error–, donde puede haber o no bits intermedios afectados. Los dos extremos afectados se consideran parte del bloque de error. Si la longitud del bloque de error es B, debe haber B o más bits no afectados entre los bloques de error.

Para el análisis de los controles de redundancia cíclica se utilizan **códigos polinómicos**, en los que el bloque a transmitir se divide por un polinomio, agregándose al bloque el resto de la división en la forma de una **secuencia de control del bloque** (FCS, *frame check sequence*), conocido normalmente como **dígitos de CRC**. El bloque se transmite (o almacena) juntamente con los dígitos de CRC. Una vez recibido el bloque, el receptor realiza el mismo cálculo, utilizando el mismo polinomio, por lo que si los restos coinciden no existen errores detectables. Pueden existir errores indetectables, por lo que el objetivo de la creación de un código CRC es la selección de un polinomio que cubra los errores estadísticamente probables en un modelo de fallas establecido.

La solución básica se inicia con la transmisión de un mensaje de k bits, $M(x)$, al que se le agregan n ceros, en el que n es el grado del **polinomio generador** $G(x)$, siendo $k > n$. Esta forma de $M(x)$ así extendida se divide por $G(x)$, utilizando aritmética de módulo 2 (en la que se descartan los bits de arrastre en sumas y restas). El resto $R(x)$, que no tiene más de n bits, determina los dígitos de CRC para $M(x)$.

Como ejemplo, considérese la transmisión del bloque siguiente:

$M(x) = 1101011011$

Asimismo, considérese, un polinomio generador $G(x) = x^4 + x + 1$. El **grado** de $G(x)$ (el exponente más alto) es 4, con lo que corresponde agregar cuatro ceros a $M(x)$ para formar el dividendo del cálculo.

El divisor es 10011, lo que corresponde a los coeficientes de $G(x)$ escrito como:

$$G(x) = 1 \cdot x^4 + 0 \cdot x^3 + 0 \cdot x^2 + 1 \cdot x^1 + 1 \cdot x^0$$

Nótese que $G(x)$ tiene grado $n = 4$ y que hay en total $n + 1 = 5$ coeficientes. Los dígitos de CRC se calculan de acuerdo con lo que se muestra en la figura 9.14. El dividendo se divide por el divisor (10011), pero las magnitudes del dividendo y del divisor no juegan papel alguno en la determinación de si el divisor "cabe" en el dividendo en la posición de un dígito particular. Todo lo que importa es que la cantidad de bits del divisor (que no tiene ceros iniciales) coincida con la cantidad de bits del dividendo (que tampoco tiene ceros iniciales en la posición verificada). Nótese que en la resta módulo 2 se opera sin arrastres y que se llega al mismo resultado si se realiza una operación XOR bit a bit entre divisor y dividendo.

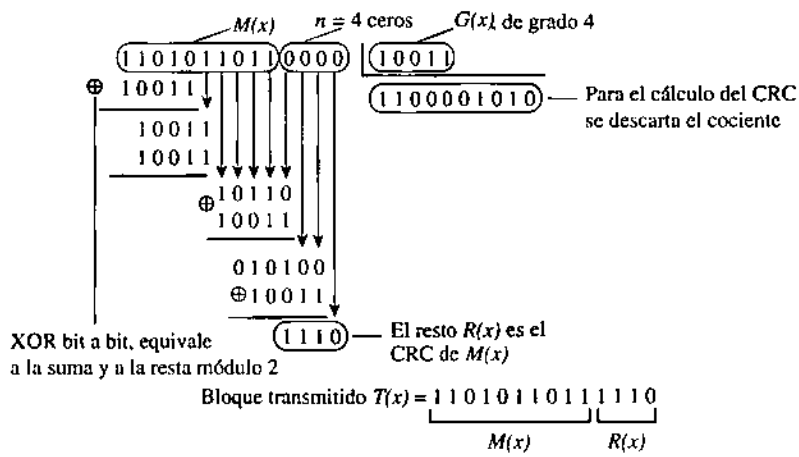


Figura 9.14 • Cálculo de los dígitos de CRC.

Supóngase ahora que el bloque transmitido $T(x) = M(x) + R(x)$ se corrompe durante la transmisión. El receptor necesita detectar este evento. El receptor divide el bloque recibido por $G(x)$. Todos aquellos errores que no contengan a $G(x)$ como factor se podrán detectar debido a la ocurrencia de un resto no nulo en cada uno de esos casos. Esto significa que, en tanto los unos de la palabra 10011 no coincidan con las posiciones de los errores en el bloque recibido, se podrán capturar todos los errores. En general, un código polinómico de grado n puede detectar todos los errores grupales de longitud $\leq n$.

Entre los polinomios de uso habitual que aseguran una buena protección contra errores se pueden mencionar los códigos:

$$\text{CRC-16} = x^{16} + x^{15} + x^{1} + x^{2} + 1$$

$$\text{CRC-CCITT} = x^{16} + x^{12} + x^{5} + 1$$

$$\text{CRC-32} = x^{32} + x^{26} + x^{23} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

El análisis profundo de los códigos CRC va más allá de los objetivos de este texto, por lo que se recomienda al lector interesado referirse al texto de R. N. Hamming para más detalles.

Ejemplo: corrección de dos errores

Se trata de determinar cuántos bits de paridad se requieren para un código basado en el código ASCII que pueda corregir dos errores. Cada carácter ASCII contiene $k = 7$ bits, a los que se deben agregar r bits de verificación en cada palabra del código. Para cada una de las palabras del código ASCII existen $k + r$ posibles combinaciones con un error, $[(k + r)(k + r - 1)]/2$ posibles combinaciones con dos errores y una combinación sin errores. Existen 2^{k+r} combinaciones posibles y, por consiguiente, se debe cumplir la siguiente relación matemática:

$$2^{k} \cdot \left[(k+r) + \frac{(k+r)(k+r-1)}{2} + 1\right] \leq 2^{k+r}$$

- 2^k: Cantidad de palabras en el código original
- $(k + r)$: Cantidad de errores simples
- $\frac{(k+r)(k+r-1)}{2}$: Cantidad de errores dobles
- 1: Palabras del código sin error
- 2^{k+r}: Cantidad de posibles combinaciones binarias

Simplificando, cuando $k = 7$, se obtiene $r^2 + 15r + 58 \leq 2r + 1$, expresión que se satisface con $r = 7$.

Dado que para corregir p errores se requiere una distancia mínima de $2p + 1$, la distancia de Hamming de un código que permita corregir dos errores debe ser al menos 5. Si se usa la misma codificación para detectar en lugar de corregir, se tiene $p + 1 = 5$, y dado que la detección de p errores requiere una distancia de $p + 1$, este código puede detectar (sin corregir) hasta $p = 4$ errores. •

9.5 Arquitectura de redes: Internet

En los primeros días de la computación, las computadoras eran instalaciones centralizadas que contenían la mayoría o la totalidad de los recursos usados por las poblaciones a las que servían. La información se transfería entre computadoras a través de medios adecuados (tarjetas de papel perforado, cintas de papel, cintas magnéticas, discos magnéticos) transportados en forma manual por un operador.

A medida que se produjo el crecimiento de la cantidad de computadoras, y que las caídas de precios del hardware implicaron un traslado de los costos hacia la mano de obra, resultó más económico vincular las computadoras directamente para poder compartir los recursos. Este es el concepto de las redes. Con anterioridad se han analizado las redes de área local en el contexto del modelo tradicional de siete capas ISO. Aquí se procederá a realizar un análisis más profundo de los aspectos arquitectónicos de las redes de computadoras, en el contexto del modelo de Internet.

9.5.1 El modelo Internet

En un sistema de telecomunicaciones pueden existir una gran cantidad de orígenes y una gran cantidad de destinos. Como ejemplo de este tipo de comunicación se puede plantear una red telefónica de larga distancia. Por cada teléfono al que se puede llegar desde otro teléfono, debe haber un camino, o **canal**, entre cada origen y destino. Si en la ciudad de Nueva York existen 10^7 teléfonos y otros 10^7 en Chicago, para que cada persona de una ciudad pueda llamar a cualquier otra persona en la otra ciudad harían falta $10^7 \times 10^7 = 10^{14}$ canales entre ambas ciudades. Afortunadamente, no todos los habitantes de Nueva York quieren hablar con todos los habitantes de Chicago en forma simultánea, por lo que todos los teléfonos de esas ciudades pueden compartir una cantidad de canales menor que la calculada. Por otra parte, puede haber al menos una línea desde cada teléfono a la oficina central de la compañía telefónica, y deben existir suficientes líneas entre las oficinas centrales como para manejar la cantidad máxima de conversaciones mantenidas en forma simultánea.

Todo lo que se requiere para conectar las dos ciudades es un pequeño número de conexiones físicas, en el orden de algunas pocas hasta algunos miles, dependiendo de si se usa fibra o cable, debido a que nunca sucede que cada uno de los habitantes de una ciudad desea llamar simultáneamente a algún habitante de la otra ciudad. La capacidad de transporte de información (**ancho de banda**) se comparte entre todos los usuarios de modo de lograr una importante reducción en el costo. No obstante, se debe crear un mecanismo de control que permita compartir adecuadamente el ancho de banda.

La formación de capas en el conjunto de protocolos TCP/IP

Se suele denominar conjunto de redes (*internet*) a una interconexión de redes individuales, de las cuales Internet es la más difundida. Internet utiliza el protocolo TCP/IP y direcciones IP en lo que se conoce como el conjunto de protocolos TCP/IP (que se desarrollará más adelante). En Internet, el modelo de siete capas de la OSI ha sido ligeramente simplificado. En este caso puede pensarse como un modelo de cuatro capas, el que se ilustra en la pila de protocolos de la figura 9.15. Al fondo de la pila se encuentra la capa de enlace, la que está compuesta a su vez por las subcapas física (PHY) y de control de acceso al medio (MAC, *medium access control*). La capa de enlace resuelve la contención del medio cuando más de un dispositivo requiere transmitir, administra la agrupación lógica de los bits en tramas e implementa la protección contra errores.

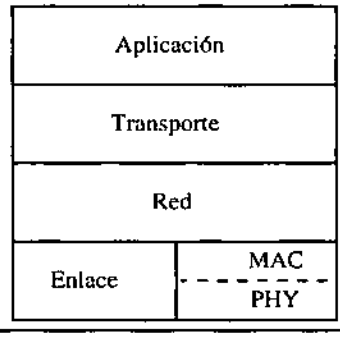


Figura 9.15 • Pila de protocolos de Internet.

La capa de enlace es responsable de obtener un conjunto de bits desde una máquina conectada directamente a otra máquina. Esto funciona bien para comunicaciones punto a punto entre dos procesos complementarios ubicados en diferentes máquinas. No obstante, con el objeto de permitir que varios procesos compartan el mismo enlace, se requiere un protocolo que determine qué información va a qué proceso. Esto es responsabilidad de la capa de red, la que se implementa con el protocolo Internet (IP, *Internet Protocol*).

La capa de red administra comunicaciones del tipo denominado salto a salto (*hop by hop*). La capa de transporte maneja comunicaciones de extremo a extremo, en las que puede haber una cantidad de sistemas intermedios entre el transmisor y el receptor. La capa de transporte se maneja con las retransmisiones (debidas a errores, a paquetes perdidos por congestiones, etc.), con la secuencia (los paquetes pueden arribar en forma desordenada), con el control de flujo (la aplicación de presión al emisor para liberar congestiones) y la protección contra errores (la capa de enlace no realiza suficiente protección de errores por sí misma). En Internet, la capa de transporte se implementa con un protocolo de control de transmisión (TCP, *Transmisión Control Protocol*). La combinación TCP/IP en las capas de transporte y de red es la serie de protocolos predominante en Internet. En las capas de enlace y de aplicación pueden usarse otros protocolos apropiados; también existen otros protocolos que se usan en las capas de red y de transporte.

En la capa de aplicación, un proceso puede intercambiar información con otro proceso que se encuentre en cualquier lugar de Internet, y puede tratar a la conexión como si fuera un archivo en el sistema local, leyendo y escribiendo bytes con las llamadas al sistema que son comunes a los procesos de lectura y escritura, implementadas frecuentemente como **conexiones** (*sockets*) que representan trayectos hacia la red a través del sistema operativo.

Direcciones de Internet

Cada interfaz ubicada sobre Internet tiene una dirección IP única. La versión 4 del protocolo IP, conocida como IPv4, sigue utilizándose ampliamente aun cuando está siendo reemplazada por IPv6. Esta última utiliza direcciones cuyo tamaño es cuatro veces mayor que el anterior, y tiene varias mejoras y simplificaciones con respecto a IPv4. Un ejemplo de dirección IPv4, representada en notación decimal, sería el siguiente:

```
165.230.140.67
```

Cada número delimitado por un punto es un byte sin signo, correspondiente al rango entre 0 y 255. La palabra binaria equivalente a la dirección IPv4 anterior es, entonces:

```
10100101.11100110.10001100.1000011
```

Los bits más significativos determinan la clase asignada a la dirección. La figura 9.16 detalla las cinco clases definidas en IPv4. La clase A tiene 7 bits de identificación de red (ID) y 24 bits para la identificación de la computadora. Entonces, puede haber, a lo sumo, 2^7 redes de clase A y 2^{24} computadoras en cada red de clase A. Sin embargo, algunas de estas direcciones están reservadas, por lo que la cantidad de direcciones que pueden asignarse a computadoras dentro de cada red es menor que la cantidad de direcciones posibles.

Las direcciones de clase B utilizan 14 bits para la identificación de la red y 16 bits para la identificación de la computadora. Las direcciones de clase C, a su vez, usan 21 bits para la identificación de la red y 8 bits para la identificación de la computadora. Las direcciones de clase D se reservan para grupos de **multidifusión** (*multicast*), a los cuales puede suscribirse cualquier sistema final con direcciones de clase A, B o C para poder recibir todo el tráfico de red organizado para ese grupo. Este mecanismo resulta eficiente para el envío de un mismo paquete a múltiples suscriptores, sin inundar la red con transmisiones y sin que el transmisor necesite llevar el control de la lista actualizada de suscriptores. Las direcciones de clase E no se utilizan.

Se prevé que las disponibilidades de direcciones según IPv4 se agoten en los primeros años de este siglo, por lo cual es importante la adopción de IPv6 a la brevedad. Ya existen muchas redes que reutilizan direcciones IP usadas en algún otro lado (por medio de un protocolo que permite compartir direcciones IP), en tanto que otras asignan direcciones IP solo durante la duración de una sesión (por ejemplo, en el caso de una línea de conexión telefónica a través de un módem).

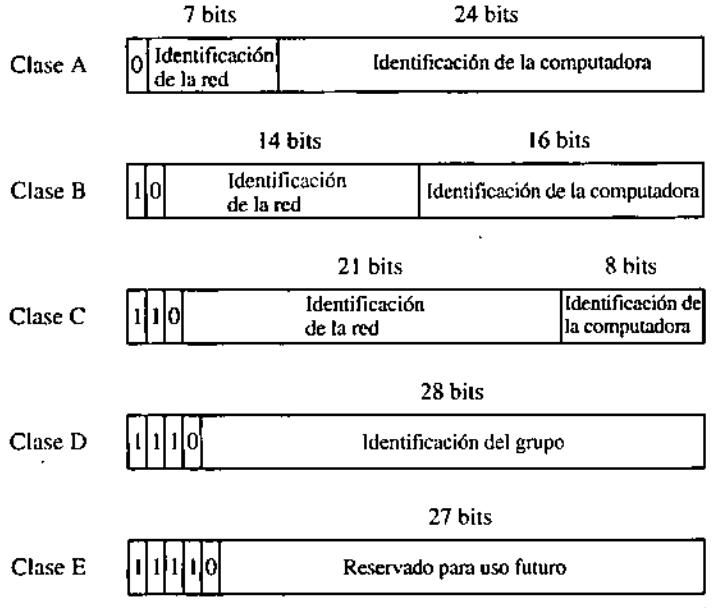


Figura 9.16 • Las cinco clases de direcciones IPV4.

Puertos

Hablando ligeramente, un **puerto** es la forma a través de la cual un proceso se hace conocer al mundo. El proceso de origen de identifica a través de un número de puerto, en tanto que un proceso de destino se identifica por otro número de puerto. Hablando rigurosamente, un puerto identifica un punto de entrada a la red por parte de un proceso. Los puertos 0-1023 son **puertos estándar** en los procesos de servidores. Por ejemplo, el puerto `telnet` es 23. En un sistema Unix, el comando

```
% telnet cereal.rutgers.edu 23
```

conectará al usuario al sistema `cereal.rutgers.edu`. Si no se coloca el `23` en la línea de comandos, se lo da por supuesto. Si se reemplaza el `23` con cualquier otro número de puerto, como el `13` para el servidor original, se producirá un acceso a un proceso diferente, con lo que la conducta resultante será distinta.

Encapsulado

A medida que la información de la red pasa a través de las capas de red, se **encapsula**, como se muestra en la figura 9.17. La información del usuario se envía a la red usando llamadas al sistema similares a las que se utilizarían para la lectura y escritura de archivos. La capa de aplicación envía la información del usuario a la capa de transporte, en la que

el sistema operativo agrega un encabezado TCP que identifica los puertos de origen y destino, lo que forma un segmento TCP. El segmento TCP se transfiere a la capa de red, en la que se reempaqueta el segmento en conjuntos de datos (datagramas), cada uno de los cuales tiene un encabezamiento IP que identifica los sistemas de origen y destino. Estos conjuntos de datos se envían a la capa de enlace, en la que los datagramas se encapsulan en bloques Ethernet (en este ejemplo). En el sistema receptor se produce un proceso inverso al descripto.

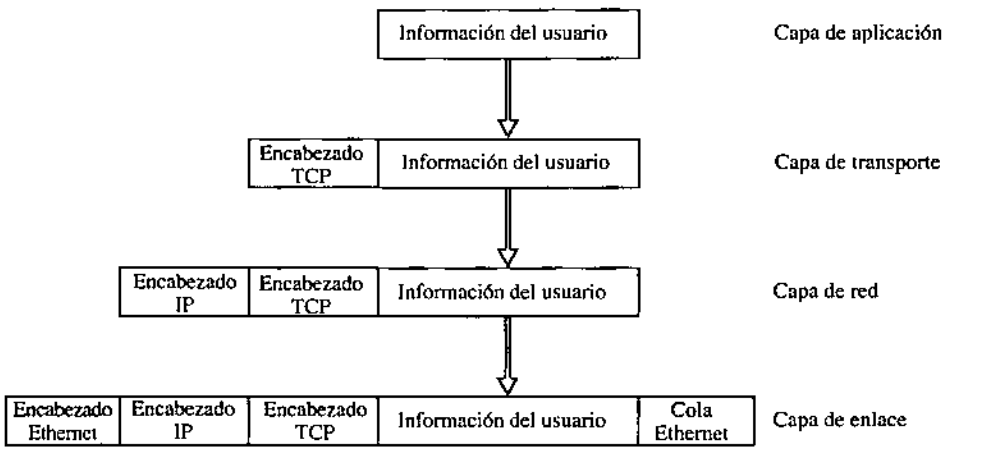


Figura 9.17 • Encapsulado en el conjunto de protocolos TCP/IP.

Un segmento TCP simple puede descomponerse en una cantidad de datagramas IP, los que se encaminan a través de Internet en forma independiente unos de otros. Cada datagrama IP contiene las direcciones IP de origen y destino (en el encabezado IP), los puertos de origen y destino (en el encabezado TCP) y el protocolo en la capa siguiente del encapsulado (en el encabezado IP, TCP es solo uno de los protocolos usados en la capa de transporte de Internet). En conjunto, estos cinco parámetros definen unívocamente cada datagrama IP en toda su travesía por la red, lo que permite asegurar que los datagramas lleguen al proceso receptor que corresponda.

El sistema de nombre de dominio

Los sistemas de nombre de dominio (DNS, *domain name systems*) constituyen una base de datos distribuida que establece una relación ente nombres de computadoras y direcciones IP, ofreciendo información de encaminamiento de mensajes. Por ejemplo, `cereal.rutgers.edu` se relaciona con `165.230.140.67` (y viceversa), y a los tres nombres: `internet.rutgers.edu`, `www.internet.rutgers.edu` y `mulder.rutgers.edu`, les corresponde `165.230.44.67`. La interfaz con aquellos programas que requieren el manejo de asignaciones entre nombres y direcciones es responsabilidad del sistema DNS.

Cada dominio (tal como `rutgers.edu`) tiene su propia base de datos de información, que mantiene en funcionamiento un servidor al que pueden acceder otros sistemas ubica-

dos en Internet. El acceso al sistema de nombres de dominio se realiza a través de un **resolvedor**, el que adopta la forma de rutinas de biblioteca que se enlazan en forma silenciosa en programas de alto nivel que acceden a la red.

El centro de información de red (NIC, *network information center*, también conocido como InterNIC) administra los dominios del nivel superior y delega la autoridad sobre los dominios del segundo nivel. Dentro de cada **zona** existe un administrador local que mantiene la base de datos de nombres de servidores. Debe existir un servidor de nombre principal, que carga su base de datos desde un archivo, y servidores de nombres secundarios, que reciben su información del servidor de nombre principal. Es necesario el uso de *caches*, de modo que una consulta que requiera el contacto con otros servidores no haga que las consultas futuras generen a su vez contactos adicionales con otros servidores.

La red mundial

La red mundial (WWW, *World Wide Web*), o, simplemente, la Red, está constituida por procesos clientes y servidores de red funcionando bajo el protocolo de transporte de hipertexto (HTTP, *HyperText Transport Protocol*) en la capa de aplicación de Internet. Dado que en el uso diario se confunden los términos, es importante tener en cuenta que la Red está construida por encima de Internet; la Red no es en sí Internet.

En 1989, Tim Berners-Lee, en CERN (una empresa europea de física de alta energía) desarrolló una red basada en texto, para el intercambio de documentos técnicos entre colegas. En febrero de 1993, el NCSA (National Center for Supercomputing Applications) de la Universidad de Illinois, en Urbana-Champaign, emitió una versión gráfica de Mosaic, programa observador (*browser*) de la Red, así como de un servidor HTTP, ambos gratuitos, y ese fue el origen de la explosión que llevó a la Red al lugar en que hoy se encuentra.

9.5.2 Revisión de puentes, encaminadores y conmutadores

Un **nodo** (*hub*) es un punto de conexión central para sistemas finales. Cuando el sistema final es otro nodo, el nodo se conoce como **puente** (*bridge*). Un nodo simplemente copia paquetes desde una interfaz de red hacia todas las demás, tal como se observa en la configuración de la figura 9.18.a. En la actualidad, los nodos y puentes tienen una inteligencia modesta, por medio de la cual pueden aislar colisiones en enlaces de una red única (esto es, si se produce una colisión entre dos paquetes en una zona de la red, lo que representa una condición normal pero no deseada, la señal que colisiona no se propaga a los restantes vínculos de la red); además, puede limitar el envío de ciertos tipos de tráfico a las restantes interfaces.

Un **encaminador** (*router*) conecta una red con otra (figura 9.18b) y toma decisiones acerca del envío de paquetes a través de sus fronteras. Por definición, el encaminador tiene más de una interfaz de red y encamina paquetes entre interfaces. Los protocolos de red usados a uno y otro lado del encaminador pueden ser diferentes.

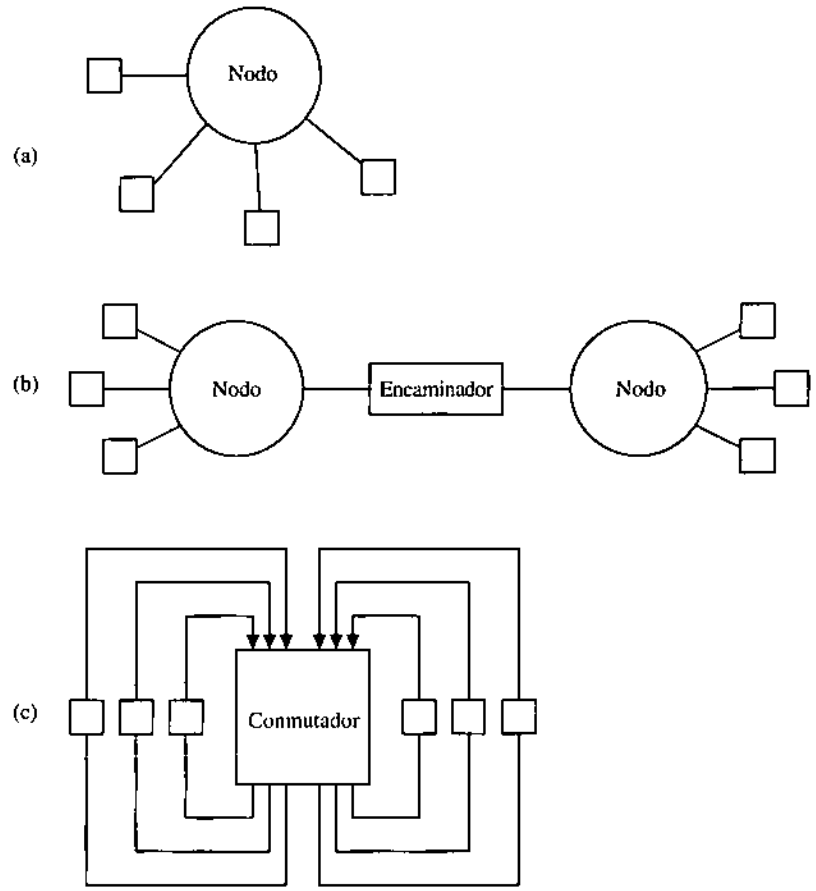


Figura 9.18 • Configuraciones de (a) un nodo; (b) un puente; (c) un conmutador.

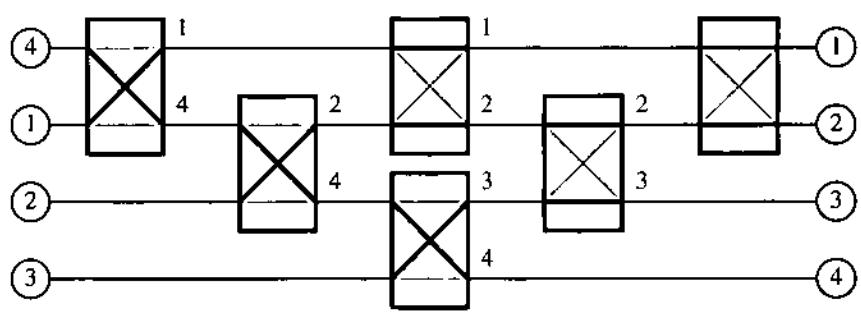


Figura 9.19 • Un conmutador de 4 x 4 con encaminamiento automático que usa un algoritmo de ordenamiento por burbujeo.

Un encaminador orienta paquetes sobre la base de un protocolo, en tanto que un **multiplexor** (*switch*) los encamina sobre la base de su dirección de destino. Un conmutador es un nodo de alta velocidad, sin ningún ancho de banda compartido, según se ilustra en la figura 9.18c. Un multiplexor no presenta conflictos de acceso a medios.

En la sección 10.9.2 se analiza un ejemplo de multiplexor, en el cual un controlador externo determina los trayectos entre origen y destino. Se puede plantear una mejora del sistema sobre la base de una **red de encaminamiento automático** (*self routing network*), que determina las conexiones entre origen y destino al vuelo, sobre la base de las direcciones de destino que se encuentran en los encabezados de los paquetes que viajan por la red.

Como ejemplo, considérese el diseño de un conmutador de encaminamiento automático de 4 entradas y 4 salidas. Se puede diseñar este elemento mediante el algoritmo de ordenamiento por **burbujeo**, en el que se colocan en el nivel superior aquellos paquetes que tengan las direcciones menores. En este método se realizan intercambios entre pares de elementos, comenzando desde el nivel superior y trabajando hacia el inferior, dejando caer hasta el fondo en cada paso el paquete con la dirección más alta. En un caso de n canales, se requieren $n(n\text{-}1)/2$ comparaciones. Para este caso, en que $n = 4$, resulta necesario realizar $4(4\text{-}1)/2 = 6$ comparaciones, lo que significa que el conmutador debe tener 6 cajas comparadoras.

El conmutador de cuatro entradas y cuatro salidas, de encaminamiento automático, se muestra en la figura 9.19. Los paquetes ingresan desordenados por la izquierda y emergen ordenados por su dirección de destino sobre la derecha del sistema. (El problema A.28 permite ver el diseño de una de estas cajas comparadoras.)

9.6 Estudio de un caso: modo asincrónico de transferencia

Históricamente, han existido redes que transportan diferentes tipos de información:

- Télex (las viejas teleimpresoras, utilizadas en agencias de noticias, cotizaciones de bolsa, etcétera).
- Telefonía, a través de la red pública conmutada (PSTN, *public switched telephone network*).
- Datos, a través de las redes de datos de conmutación de paquetes (PSDN, *packet switched data network*).
- Televisión: a través de (1) antenas fijas en tierra; (2) antenas de televisión comunitarias (televisión por cable, conocida como CATV, *community antennae* TV); (3) satélites.
- Información local, a través de redes de área local.

Cada red se planifica, se dimensiona, se desarrolla y se opera en forma separada. Esta especialización da por resultado un conjunto de redes mundiales independientes entre sí. El

ancho de banda no se comparte entre las redes, y hasta ocurre que en un horario central, mientras aumenta el tráfico de video, el tráfico telefónico disminuye apreciablemente. Aun habiendo disponibilidad de programas de video en un canal en particular, en cualquier horario del día, pueden usarse programas locales, más económicos, fuera de las horas pico, reduciendo así la necesidad de tráfico de video vía satélite o a través de redes cableadas de larga distancia.

Cada tipo de servicio tiene que manejar el tráfico pico y el tráfico por bloques por las suyas, si bien sería más económico compartir las redes en tanto los picos y los bloques no coincidan. Las **redes de servicios digitales integradas** (ISDN, *Integrated Services Digital Network*) son un intento de compartir recursos. Las mismas pueden adoptar dos formatos. Las redes ISDN de **banda angosta** (NISDN, *Narrowband* ISDN) se diseñan para la conmutación telefónica de 64 Kbps. Estas redes permiten el transporte de tráfico de voz y datos sobre la misma red. Las redes ISDN de **banda ancha** (BISDN, *Broadband* ISDN) no solo manejan tráfico de voz y de datos, sino también de video.

Las redes NISDN utilizan una **interfaz de velocidad básica** (BRI, *basic rate interface*), la que ofrece dos canales B a 64 Kbps para la información del usuario y un canal de datos D a 16 Kbps utilizado para control y señalización. Cuando se toma en cuenta el agregado del nivel superior, se obtiene una velocidad total de 192 Kbps. Para el usuario, la velocidad máxima es de 128 Kbps cuando se juntan los dos canales B en un canal único.

La **interfaz de velocidad primaria** (PRI, *primary rate interface*) ISDN, que ofrece 23 canales B y un canal D a 64 Kbps, se conoce habitualmente con el nombre de "línea T1".

Las líneas ISDN pueden ser alquiladas a las compañías de telecomunicaciones y configuradas para implementar redes privadas. Como regla empírica general, el valor del alquiler mensual de diez líneas de 64 Kbps equivale al costo de alquilar una línea T1, que puede manejar 23 líneas de 64 Kbps. Existe un problema económico relacionado con ISDN, que surge debido a que las velocidades que soportan son dependientes del servicio (NISDN es un ejemplo de un servicio) en lugar de ser dependientes del tráfico, lo que sería necesario en las redes modernas. Se debe requerir que las velocidades ofrecidas resulten múltiplos pares de los canales B y D, ya que de no ser así no harán uso eficiente del ancho de banda disponible. Dado que, generalmente, no se conocen las velocidades que podrán necesitar los futuros servicios, se debe requerir un transporte de datos independiente del servicio, con el objeto de lograr que la red pueda extenderse. El objetivo es obtener una red única que pueda servir a todos, momento en el cual hace su aparición el **modo asincrónico de transferencia** (ATM, *asynchronous transfer mode*).

9.6.1 Transferencia sincrónica contra transferencia asincrónica

En el **modo de transferencia sincrónica**, también conocida como **multiplexado por división del tiempo** (TDM, *time division multiplexing*), el conjunto de datos se descompone en intervalos de tiempo que se asignan a los distintos canales en forma secuencial, continua y cerrada. La figura 9.20a ilustra el funcionamiento del sistema sobre cuatro ca-

nales. Cada una de las estaciones puede transmitir solo durante el intervalo de tiempo que le fuera preasignado. Mientras una estación espera que le llegue el turno, pueden sucederse intervalos de tiempo no utilizados (en la línea conmutada de telefonía pública, por ejemplo, el intervalo de tiempo es de 125 μs, lo que permite 8.000 muestras por segundo a 8 bits por muestra, y da por resultado 64 Kbps en una línea con calidad de voz).

La figura 9.20b muestra el modo asincrónico de transferencia; este puede seguir utilizando los mismos 125 μs por ventana, pero con el criterio de que cada estación pueda utilizar cualquier intervalo, lo que hace que la red funcione en forma más eficiente. La desventaja en la operación de una red que funciona en modo asincrónico es su complejidad mucho mayor que la de una operación simple de multiplexado por división de tiempos.

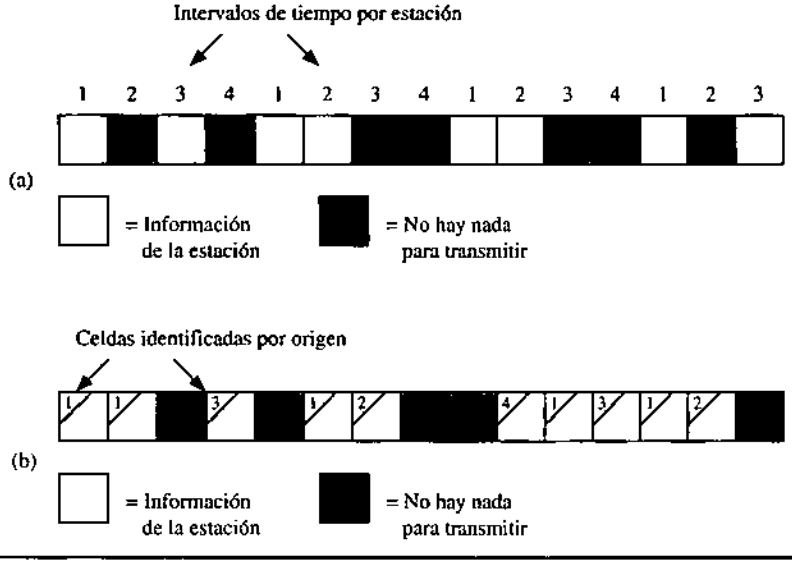


Figura 9.20 • (a) Multiplexado por división del tiempo versus (b) modo asincrónico de transferencia.

9.6.2 ¿Qué es el modo asincrónico de transferencia?

El modo asincrónico de transferencia es una combinación de hardware con un conjunto de protocolos que entrega un ancho de banda garantizado, con un tiempo de latencia bajo y limitado. Las dos áreas de progreso que hacen posible la metodología ATM son:

1. Tecnología. La velocidad y la densidad de los circuitos integrados en elevada escala de integración (VLSI) permiten introducir la funcionalidad de las redes directamente en los sistemas finales, en vez de instalarla en los elementos componentes intermedios. Por ejemplo, la detección de errores solo se implementa en los extremos, en vez de realizarse en cada dispositivo intermedio de encaminamiento, como suce-

de comúnmente en Internet. El desarrollo de fibras ópticas de gran ancho de banda con una tasa de error muy baja es una tecnología que permite aceptar esta solución como adecuada.

2. Sistemas. Multiplexores de paquetes de alta velocidad, que introducen bajos tiempos de latencia en las comunicaciones punto a punto y que permiten el manejo de servicios en tiempo real.

Las velocidades típicas de los sistemas ATM están en el rango de 1 Gbps a 2,5 Gbps, con un retardo promedio de solo 450 μs para cada conmutador que maneja un paquete ATM.

9.6.3 Arquitectura de una red ATM

Antes de que se produzca una transferencia en modo asincrónico, se debe establecer una conexión a través del contacto con un **punto de control de señalización** (SCP, *signaling control point*), dispositivo de la red que tiene autoridad como para configurar los conmutadores para la transferencia requerida. Todos los paquetes encuentran como restricción la obligación de seguir el trayecto establecido por el punto de control, en consecuencia, todos los paquetes arriban en orden.

El trabajo del conmutador es muy simple. Lo único que hace es verificar la dirección de destino que figura en el encabezado, tras lo cual envía el paquete por el camino también indicado en el encabezado. De acuerdo con que lo que se analiza a continuación, además se coloca en el encabezado una nueva dirección de destino.

Todos los paquetes de un sistema ATM tienen un mismo tamaño fijo de 53 bytes, lo que se ilustra en la figura 9.21. El paquete que el usuario final crea para su inyección en la red se ilustra en la figura 9.21a, y se encuentra configurado en el formato de **interfaz entre usuario y red** conocido como UNI (*user to network interface*). A partir de ese punto, el formato utilizado es el que se conoce como formato de **interfaz entre redes** (NNI, *network to network interface*), que se muestra en la figura 9.21b.

El campo GFC de control genérico de flujo se utiliza en la frontera de la red para determinar las políticas que autorizan el ingreso de los paquetes a la red. Una vez que el paquete ya está en la red, el campo GFC del formato UNI se vuelve innecesario, por lo que se lo combina con el campo identificador de trayecto virtual (VPI) para formar un campo VPI de 12 bits en el formato NNI. El campo VPI puede entenderse como la identificación de un domicilio particular en un cable que transporta señales de canales de televisión por cable, mientras que el campo identificador de canal virtual (VCI) identifica el canal específico. El identificador de tipo de carga (PTI, *payload type identifier*) determina si el campo de datos transporta información del usuario o información de la red u otra. El bit de prioridad de pérdida de celda (CLP, *cell loss priority*) determina si una celda en particular puede descartarse en momentos de congestión. El control de error del encabezado (HEC, *header error control*) es un control cíclico de redundancia sobre el encabezado. A continuación, se tiene un campo de datos de 48 bytes.

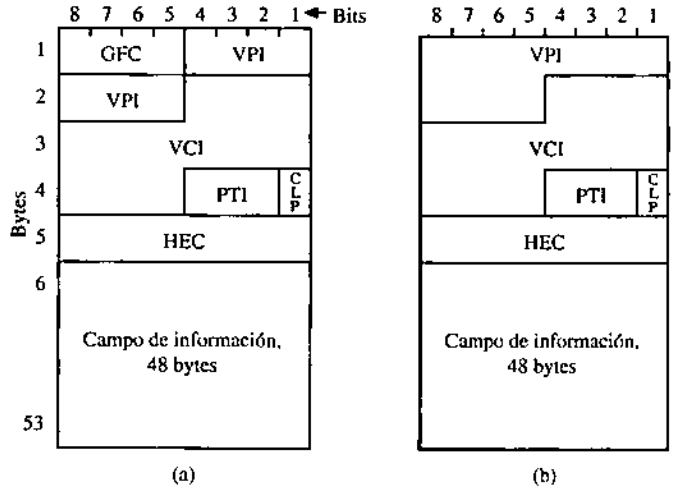


Figura 9.21 • Formato de un paquete ATM: (a) formato de interfaz entre usuario y red (UNI); (b) formato de interfaz entre redes (NNI).

La figura 9.22 muestra una red ATM simple constituida por tres conmutadores. Cada uno de ellos tiene una tabla de encaminamientos simétrica que puede manejarse en cualquier sentido. Por ejemplo, el conmutador 1 puede recibir un paquete entrante (por la izquierda) que tiene un campo VPI de 7. Tras modificar el campo VPI a 5, el paquete se envía hacia la derecha. De la misma forma, un paquete que ingresa al conmutador 1 desde la derecha con un campo VPI de 5 se envía hacia la izquierda con un campo VPI de 7. Este proceso se denomina conmutación virtual de trayecto, debido a que solo se utiliza el campo VPI para el encaminamiento de los datos. El campo VCI permanece inalterado. También puede haber conmutación virtual de canales; en este caso se encaminan los datos sobre la base del campo VCI y es el campo VPI el que permanece inalterado.

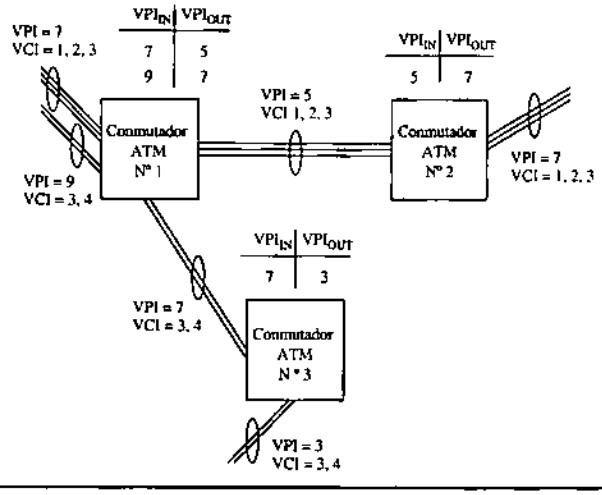


Figura 9.22 • Conmutación VPI en una red de transferencia asincrónica (adaptado de Martin de Prycker).

9.6.4 Perspectivas de ATM

Si bien se observa que el esquema de transferencia asincrónica de ATM promete grandes cosas para una red extensible que brinda servicios a las necesidades de muchos usuarios, las economías de escala juegan a favor de redes como Ethernet para los sistemas de los usuarios finales. ATM aparece fundamentalmente en redes principales, que utilizan concentradores y multiplexores para conectarse con otras redes menores, pero la conexión se establece en los límites de la red principal. El modo asincrónico de transferencia continúa con su penetración en redes principales y también en los sistemas de escritorio, aun cuando esta es la excepción más que la regla. Solo el tiempo podrá decir cuál será el grado de penetración de la tecnología ATM en sistemas de escritorio.

Resumen

Las comunicaciones implican la transferencia de información entre sistemas. Como regla, los datos se transfieren en forma serie, bit por bit, debido a que en redes de alta velocidad los retardos de los tiempos de comunicación predominan por sobre los tiempos de transferencia. No obstante, los esquemas de modulación permiten la codificación de varios bits en una única muestra, como en el caso de los dibits. La posibilidad de elegir el esquema de modulación afecta a la introducción de errores en el proceso de transmisión. La detección y la corrección de errores se hacen posibles a través de la redundancia, lo que implica mayor cantidad de palabras binarias que la cantidad de palabras válidas. Si el patrón binario de error no tiene una única palabra binaria válida adyacente, se pueden detectar errores pero no corregirlos. Si cada palabra inválida se encuentra adyacente a una única palabra binaria valida, es también posible la corrección de errores.

Las redes de área local (LAN) manejan la complejidad a través del uso de capas que se basan en el modelo OSI. En estos días, las redes de área local se encuentran conectadas con redes de área extendida (WAN), de las cuales Internet es la más notable. Internet se basa en el conjunto de protocolos TCP/IP. Los datos del usuario se encapsulan en las capas de aplicación, transporte, red y enlace, se envían a través de Internet y se recuperan en el sistema receptor. A medida que la información viaja por Internet, recorre distintos medios de transmisión, los que pueden diferir en sus prestaciones de ancho de banda y distancia.

Para lectura posterior

C. E. Shannon desarrolló las bases de la teoría moderna de las telecomunicaciones. R. Needleman y P. Schnaidt, en sus respectivas obras, hacen un análisis profundo de las redes de área local de acuerdo con el modelo OSI. A. Tanenbaum (1996) es una buena referencia sobre el tema de comunicación por redes, en general. F. Halsall ofrece un tratamiento profundo y claro referido a los

tipos de medios usados en redes. M. de Prycker analiza en forma profunda el concepto de ATM y sus características.

A. Tanenbaum (1999) y W. Stallings explican claramente la codificación Hamming. Tanto R. W. Hamming como W. Peterson y E. J. Weldon tratan con gran detalle los códigos correctores de error.

Halsall, F., *Data Communications, Computer Networks and Open Systems*, 4ª ed., Addison Wesley, 1996. (Traducción al español disponible: *Comunicación de datos, redes de computadoras y sistemas abiertos*, 4ª ed., Addison Wesley, 1998.)

Hamming, R. W., *Coding and Information Theory*, 2ª ed., Prentice Hall, 1986.

Needleman, R., *Understanding Networks*, Simon and Schuster, 1990.

Peterson, W. Wesley y E. J. Weldon Jr., *Error-Correcting Codes*, 2ª ed., MIT Press, 1972.

de Prycker, Martin, *Asynchronous Transfer Mode: Solution for Broadband ISDN*, 2ª ed., Ellis Horwood, 1993.

Schnaidt, P., *LAN Tutorial*, Miller Freeman Publications, 1990.

Shannon, C. E., "A mathematical theory of communication", en: *Bell System Technical Journal*, vol. 27, julio y octubre de 1948, p.p. 379-423 y 623-656.

Stallings, W., *Computer Organization and Architecture: Designing for Perfomance*, 4ª ed., Prentice Hall, 1996. (Traducción al español disponible: *Organización y arquitectura de computadores*, 5ª ed., Prentice Hall, 2000.)

Tanenbaum, A., *Computer Networks*, 3ª ed., Prentice Hall, 1996. (Traducción al español disponible: *Redes de computadoras*, 3ª ed., Prentice Hall, 1997.)

Tanenbaum, A., *Structured Computer Organization*, 4ª ed., Prentice Hall, 1999. (Traducción al español disponible: *Organización de computadoras*, 4ª ed., Prentice Hall, 2000.)

Problemas

9.1 ¿Cuál es la distancia Hamming del código ASCII corrector de un error, analizado en la sección 9.4.2?

9.2 Implementar la codificación ASCII para corregir un error sobre el carácter 'Q' usando paridad par.

9.3 Construir un código corrector de un error, con paridad par, para los puntos siguientes:

a. ¿Cuántos bits de verificación se requieren para codificar una palabra de seis bits?

b. Construir el código corrector para la palabra de seis bits 101100. Al construirlo, numerar los bits de acuerdo con el método detallado en la sección 9.4.2.

c. Un receptor recibe una palabra de dos bits codificada en un código corrector de un error, cuyo formato es 11100. ¿Cuál es la palabra de dos bits original?

d. Se recibe la palabra de 12 bits 101110011001, la que incluye codificación para corrección de un error, de paridad par. ¿Cuál fue la palabra de 12 bits realmente transmitida?

9.4 ¿Cuántos bits de verificación se requieren para implementar un código corrector de errores simples si el tamaño original de la palabra es de 1024 bits?

9.5 Construir la palabra de verificación (*checksum*) correspondiente a las letras 'V' a 'Z', codificadas en EBCDIC, utilizando paridad par para el control de redundancia vertical. No utilizar control de redundancia longitudinal.

9.6 Comparar la cantidad de bits utilizados para paridad en el código corrector de un error con el código de control de redundancia vertical, para el caso de 1024 palabras de 8 bits.

a. Calcular el número de bits de verificación generados si solo se usa un código de corrección de errores simples en forma horizontal.

b. Calcular la cantidad de bits de verificación si solo se utiliza control de redundancia vertical.

9.7 El código de corrección de errores simples analizado en la sección 9.4.2 se puede convertir en un código de detección de errores dobles (DED/SEC) por medio del agregado de un bit adicional que genera paridad par en el código corrector de un error (la que incluye el bit de paridad agregado). Explicar cómo funciona la detección de dos errores mientras se permite corregir también un solo error.

9.8 Calcular el control de redundancia cíclica para el mensaje $M(x) = 101100110$ utilizando un polinomio generador $G(x) = x^3 + x^2 + 1$.

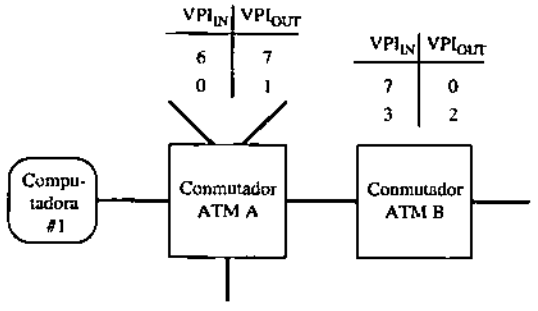
9.9 ¿Cuál es el bloque de error más largo que puede detectarse con seguridad en el algoritmo CRC-32?

9.10 ¿A qué clase IPv4 corresponde la dirección `165.230.140.67`?

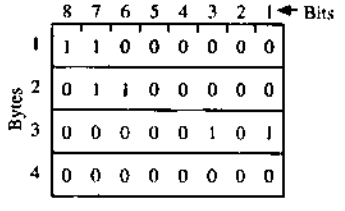
9.11 ¿Cuántas redes (no computadoras) pueden soportar las direcciones IPv4 de clases A, B, C? De otra manera, ¿cuántas direcciones de red de clases A, B y C pueden existir? No considerar las direcciones reservadas.

9.12 Los medios de red siempre transportan la información en formato serie bit por bit, y casi nunca en paralelo. Esto no significa que no se pueda transferir información a través de una red en formato paralelo por bytes o por palabras, sino, simplemente, que no existe ventaja en transferirla de esta forma. Para justificar esta afirmación, calcular el tiempo requerido para transmitir una palabra de 32 bits entre dos computadoras a través de una red de 10 m (32 pies) de longitud. La velocidad de la red es de 1 Gbps por canal. El retardo impuesto por la distancia es de 3 ns por metro. Calcular el tiempo requerido para transmitir la palabra de 32 bits utilizando un único canal (en serie, bit por bit). Calcular el tiempo necesario para transmitirla usando 32 canales (en paralelo, palabra por palabra).

9.13 Una red ATM tiene dos conmutadores A y B que conmutan solo sobre el identificador de trayecto virtual. La topología de red y las tablas de encaminamiento son las siguientes:



Se recibe un paquete en el conmutador A proveniente de la computadora 1, el que contiene un encabezado UNI de cuatro bytes como el siguiente:



Mostrar los cuatro primeros bytes del encabezado NNI de salida.

Capítulo 10

Avances en la arquitectura de computadoras

En los capítulos anteriores, al describir el ciclo de búsqueda y ejecución de instrucciones se lo planteó en la forma "buscar una instrucción, ejecutarla, buscar la instrucción siguiente, etc.". Esta descripción representa la ejecución de un programa en la forma de una progresión lineal. El hecho es que las arquitecturas actuales de los procesadores ofrecen muchas prestaciones avanzadas que van más allá de este simple paradigma. Estas prestaciones incluyen la **segmentación de instrucciones** (*pipelining*), estructura según la cual distintas instrucciones que comparten el mismo hardware pueden encontrarse simultáneamente en distintas fases de ejecución; la **ejecución superescalar**, en la que varias instrucciones se ejecutan en forma simultánea usando diferentes porciones del hardware, y donde probablemente solo algunos de los resultados aporten algo al resultado completo; las arquitecturas con **palabras de instrucción muy largas** (VLIW, *very long instruction word*), en las que cada palabra especifica varias instrucciones (de menor tamaño) que se ejecutan en forma simultánea, y el **procesamiento paralelo**, en el que se coordina la acción de varios procesadores que trabajan sobre un único problema.

Este capítulo analiza temas asociados con estas características. El análisis comienza con aquellas cuestiones que llevaron a la aparición de las computadoras con **conjuntos reducidos de instrucciones** (RISC, *reduced instruction set computer*), aportando ejemplos de las características y prestaciones de las arquitecturas RISC. A continuación, se analiza una característica avanzada que se utiliza específicamente en la arquitectura de procesadores SPARC: la **superposición de ventanas de registros**. El análisis continúa con dos características arquitectónicas importantes: la ejecución superescalar y las arquitecturas VLIW. Luego, se avanza hacia el tema del procesamiento paralelo, analizando tanto las arquitecturas paralelo como la descomposición de programas. El capítulo incluye el estudio de casos que cubren la arquitectura Merced de Intel, el procesador PowerPC 601 y un ejemplo de una arquitectura paralelo que puede encontrarse en un sistema implementado en un videojuego hogareño.

10.1 Análisis cuantitativo de la ejecución de un programa

Antes de los finales de la década del setenta, los arquitectos de computadoras aprovecharon los avances de las tecnologías de circuitos integrados para aumentar la complejidad de las instrucciones y de los modos de direccionamiento, dado que suponían obvios los beneficios resultantes de dichas mejoras. Tener instrucciones y modos de direccionamiento más complejos que los de los procesadores competidores se convirtió en una técnica de ventas efectiva. Los aumentos en la complejidad de las arquitecturas llevaron al convencimiento de que la barrera más importante para lograr un mejor rendimiento de las máquinas se encontraba en la **brecha semántica**, o sea, el salto entre los significados de las sentencias de los lenguajes de alto nivel y los de las instrucciones a nivel del lenguaje de máquina.

Desgraciadamente, a medida que los arquitectos hacían intentos por mejorar esa brecha semántica, a veces la empeoraban. La arquitectura IBM 360 tiene una instrucción MVC (mover caracteres) que copia un conjunto de hasta 256 caracteres entre dos posiciones arbitrarias. Si los sectores de origen y destino se superponen, la porción superpuesta se copia de a un byte por vez. El análisis que se lleva a cabo durante la ejecución para determinar el grado de superposición agrega un exceso de tiempo significativo al tiempo de ejecución propio de la instrucción MVC. Las determinaciones realizadas muestran que las superposiciones se producen solo algunas pocas veces y que el tamaño promedio del vector a copiar es de solo ocho bytes. En general, se obtienen mejores resultados en la ejecución cuando la instrucción MVC se ignora por completo, sintetizando su ejecución a través de instrucciones más simples. Si bien se utiliza una cantidad mayor de instrucciones si no se usa la instrucción MVC, en el promedio se requiere una menor cantidad de ciclos de reloj cuando la operación de copia se realiza sin usar la instrucción MVC.

Algunas ideas mantenidas durante largo tiempo comenzaron a cambiar en 1971, cuando Donald Knuth publicó un análisis panorámico de programas típicos escritos en lenguaje Fortran, en el que mostraba que la mayoría de las sentencias son simples asignaciones. Una investigación posterior, llevada a cabo por John Hennessy en la Universidad de Stanford y por David Patterson de la Universidad de California en Berkeley, confirmó que los compiladores casi no utilizaban las instrucciones y modos de direccionamiento más complejos. Estos investigadores popularizaron el uso del análisis de programas y de los programas evaluadores de rendimiento (*benchmark*) con el objeto de medir el impacto de la arquitectura sobre el rendimiento.

Sentencias	Tiempo medio (porcentual)
Asignaciones	47
Condicionales	23
Llamadas	15
Lazos	6
Saltos	3
Otras	7

Figura 10.1 • Frecuencia de ocurrencia de los diversos tipos de instrucciones en distintos lenguajes. Los porcentajes no suman el 100% debido al redondeo (adaptado de D. E. Knuth).

La figura 10.1, tomada de D. E. Knuth, resume la frecuencia en que se producen las instrucciones en una mezcla de programas escritos en una cantidad de lenguajes. Casi la mitad de las instrucciones son sentencias de asignación. Como dato interesante, las operaciones aritméticas y otras "más poderosas" solo suman el 7% del total de las instrucciones. Por lo tanto, si se deseara mejorar el rendimiento de una computadora, los esfuerzos se aprovecharían mejor si se optimizaran las instrucciones que influyen sobre el mayor porcentaje de tiempo de ejecución que si se enfocaran instrucciones que son implícitamente complejas en sí mismas pero que se utilizan muy rara vez.

	Cantidad de términos (porcentaje) en las asignaciones	Cantidad de variables locales (porcentaje) en los procedimientos	Cantidad de parámetros (porcentaje) en llamadas a procedimientos
0	–	22	41
1	80	17	19
2	15	20	15
3	3	14	9
4	2	8	7
≥ 5	0	20	8

Figura 10.2 • Porcentajes que ilustran la complejidad de asignaciones y llamadas a procedimientos (adaptado de A. Tanenbaum).

Las métricas relacionadas se muestran en la figura 10.2. De la figura surge que la cantidad de términos en una sentencia de asignación normalmente suele ser baja. El caso más frecuente (80%) corresponde a la sentencia de asignación de una única variable X ← Y. En un procedimiento, solo hay unas pocas variables locales, y, por otra parte, los argumentos que se le transfieren a un procedimiento son algunos pocos.

De las mediciones se puede ver que la mayoría de los programas de computadora son muy simples a nivel de las instrucciones, aun cuando podrían crearse programas más complejos. Esto significa que puede haber poco o ningún costo al aumentar la complejidad de las instrucciones.

Como experiencia desalentadora adicional, el análisis de códigos compilados mostró que normalmente los compiladores no aprovechan las instrucciones y modos de direccionamiento más complejos, que habían sido desarrollados por los diseñadores ansiosos de cerrar la brecha semántica. Una razón importante que permite justificar este fenómeno se halla en la dificultad que significa para el compilador el analisis del código con el detalle suficiente como para localizar los sectores de programa que podrían usar eficientemente estas nuevas instrucciones. Esto, a su vez, se debe a la gran diferencia entre el significado de la mayoría de las construcciones escritas en lenguaje de alto nivel y la forma en que las construcciones se expresan en lenguaje ensamblador. Esta observación y la velocidad y capacidad siempre crecientes de la tecnología de los circuitos integrados se juntaron para lograr la evolución desde las computadoras de instrucciones complejas (CISC, *complex instruction set computer*) hacia las arquitecturas RISC.

Un planteo básico de la arquitectura actual de computadoras es resolver en forma rápida el caso frecuente, y esto muy a menudo implica hacerlo simple. Dado que la sentencia de asignación se produce tan frecuentemente, los esfuerzos deberían estar concentrados en hacerla veloz (y, en consecuencia, simple). Una forma de simplificar las asignaciones consiste en hacer que todas las comunicaciones con memoria se concentren en solo dos comandos: CARGAR (*LOAD*) y ALMACENAR (*STORE*). El modelo de carga y almacenamiento es típico de las arquitecturas RISC. El concepto correspondiente ha sido analizado en el capítulo 4 con relación a las instrucciones `ld` y `st` de ARC.

Si se les restringe el acceso a memoria nada más que a las instrucciones de carga y almacenamiento, las demás instrucciones solo pueden acceder a datos almacenados en registros. Esto produce dos consecuencias, ambas buenas y malas: (1) los accesos a memoria pueden superponerse fácilmente, debido a que hay menores efectos secundarios que los que se tendrían si hubiese distintos tipos de instrucciones capaces de acceder a memoria, y (2) se requiere una gran cantidad de registros (esto parece malo, pero conviene seguir leyendo).

Un conjunto de instrucciones más simple da por resultado una CPU más simple y más pequeña, lo que libera espacio del circuito integrado que puede ser usado para otra cosa, por ejemplo, registros. Por lo tanto, la necesidad de más registros se balancea en cierto grado con el área de circuito que ha quedado recientemente liberada. El problema clave ahora consiste en determinar cómo se administran estos registros, lo que se analiza en la sección 10.4.

10.1.1 Análisis cuantitativo del rendimiento

Cuando se estima el rendimiento de una máquina, generalmente la medida más importante es el tiempo de ejecución, T. Cuando se considera el impacto de alguna mejora en el rendimiento, el efecto de la mejora se expresa habitualmente en función del **aumento de velocidad** (*speedup*) S, tomado como la relación entre el tiempo de ejecución medido sin la mejora (T_{sin}) y el tiempo de ejecución de la misma operación medido con la mejora (T_{con}):

$$S = \frac{T_{sin}}{T_{con}}$$

Por ejemplo, si el agregado de una memoria *cache* de 1 MB en un sistema de computación implica que un programa que se ejecutaba en 12 segundos pase a ejecutarse en 8 segundos, el aumento de velocidad sería de 12/8 = 1,5, o, lo que es lo mismo, un 50%. La ecuación que permite calcular el aumento de velocidad como un porcentaje directo puede representarse como:

$$S = \frac{T_{sin} - T_{con}}{T_{con}} \times 100$$

Se puede desarrollar una ecuación algo más refinada si se tienen datos acerca del período τ del reloj de la máquina, de la cantidad *CPI* de ciclos de reloj por instrucción y de la cantidad total de instrucciones *IC* ejecutadas por el programa. En este caso, el tiempo total de ejecución del programa se determina a través de:

$$T = \tau \times IC \times CPI$$

Tanto *CPI* como *IC* pueden expresarse como el promedio del conjunto de instrucciones y de la cantidad total, respectivamente, o bien pueden realizarse las sumas considerando cada tipo y cantidad de instrucciones del conjunto de instrucciones y del programa. Si se reemplaza la última instrucción en la anterior, se tiene:

$$S = \frac{IC_{sin} \times CPI_{sin} \times \tau_{sin} - IC_{con} \times CPI_{con} \times \tau_{con}}{IC_{con} \times CPI_{con} \times \tau_{con}} \times 100$$

Estas ecuaciones y otras que se derivan de ellas son útiles en el cálculo y estimación del impacto que producen sobre el rendimiento los cambios que se introducen en las instrucciones y en la arquitectura.

Ejemplo: cálculo del aumento de velocidad en un nuevo conjunto de instrucciones

Supóngase que se desea estimar el aumento de velocidad obtenido al reemplazar una CPU con un promedio de 5 ciclos de reloj por instrucción por otra cuyo promedio sea de 3,5 ciclos por instrucción, en la cual se ha incrementado el período de la señal de reloj de 100 ns a 120 ns. La ecuación anterior se convierte en:

$$S = \frac{5 \times 100 - 3{,}5 \times 120}{3{,}5 \times 120} \times 100 = 19\%$$

Así pueden estimarse los impactos sobre el rendimiento producidos por cambios en la arquitectura, sin necesidad de ejecutar programa de evaluación alguno. •

10.2 De CISC a RISC

Históricamente, cuando los tiempos de los ciclos de memoria eran muy largos y los precios de las memorias eran elevados, los programas con pocas instrucciones pero complicadas tenían ventaja sobre aquellos con instrucciones más simples pero con mayor cantidad de ins-

trucciones. Sin embargo, llegó un momento en el que las memorias se hicieron suficientemente económicas y las jerarquías de memoria se hicieron bastante veloces y grandes como para que los arquitectos de computación comenzaran a replantear esta ventaja. Una tecnología que influyó sobre este análisis fue la de la **segmentación** (*pipelining*), lo que significa mantener la estructura de la unidad de ejecución más o menos similar pero permitiendo que las diferentes instrucciones (cada una de las cuales requiere varios ciclos de reloj para su ejecución) utilicen distintas partes de la unidad de ejecución en cada ciclo de reloj. Por ejemplo, mientras una de las instrucciones puede estar accediendo a los operandos en el conjunto de registros, otra instrucción puede estar utilizando la unidad aritmético-lógica.

El concepto de segmentación será analizado con más detalle en secciones posteriores del capítulo, pero es importante tener en cuenta aquí que los diseñadores de computadoras descubrieron que las instrucciones de las arquitecturas CISC no encajaban bien en las arquitecturas segmentadas. Para que la segmentación funcione en forma efectiva, cada instrucción debe ser semejante a otras instrucciones, al menos en términos de la complejidad relativa de las instrucciones. La razón puede verse si se establece una analogía con una línea de producción que fabrica distintos modelos de un automóvil. Para lograr eficiencia, cada una de las estaciones de la línea de montaje debería realizar aproximadamente la misma cantidad y tipo de trabajo. Si la cantidad o tipo de trabajo que se realiza en cada una de las estaciones es radicalmente diferente para los distintos modelos, cada tanto se deberá producir una parada de la línea de montaje para acomodarse a los requerimientos del modelo en cuestión.

Los conjuntos de instrucciones CISC tienen como desventaja que algunas instrucciones, como los movimientos entre registros, son simples por su propia característica, en tanto que otras, como la instrucción MVC y otras similares a ella, son complejas y pueden requerir muchos más ciclos de reloj para ejecutarse.

Las claves filosóficas principales de la arquitectura RISC son las siguientes:

- La búsqueda anticipada de las instrucciones hacia una cola de instrucciones en la CPU antes de que se las necesite realmente. Como consecuencia de esto, se oculta la latencia asociada con la búsqueda de la instrucción.
- Cuando el tiempo de búsqueda de la instrucción deja de ser un problema, y con memorias baratas que permiten contener una gran cantidad de instrucciones, no hay una ventaja cierta para las instrucciones CISC. Todas las instrucciones deberían estar compuestas por secuencias de instrucciones RISC, aun cuando la cantidad de instrucciones requeridas pueda aumentar (generalmente, alrededor de 1/3 más que en la solución CISC).
- El movimiento de operandos entre registros y memoria es caro y debería ser minimizado.
- El juego de instrucciones RISC debería diseñarse con la mente puesta en una arquitectura segmentada.
- No hay razón para que las instrucciones CISC se mantengan como unidades integradas; pueden descomponerse en secuencias de instrucciones RISC más simples.

El resultado es una arquitectura RISC con características que la distinguen de las arquitecturas CISC:

- Todas las instrucciones son de longitud fija, cuyo tamaño es una palabra.
- Todas las instrucciones ejecutan operaciones simples que pueden llevarse hacia la estructura segmentada a razón de una por ciclo de reloj. Las operaciones complejas deben ser convertidas por el compilador en instrucciones simples.
- Todos los operandos deben estar almacenados en registros antes de que se pueda operar sobre ellos. Existe un conjunto separado de instrucciones para acceso a memoria: LOAD y STORE. Esto recibe el nombre de arquitectura de carga y almacenamiento.
- Los modos de direccionamiento se limitan a los más simples. Los cálculos más complejos de direccionamiento se construyen usando secuencias de operaciones simples.
- Deberá haber una gran cantidad de registros de uso general para realizar operaciones aritméticas de modo tal que las variables temporarias puedan almacenarse en registros en lugar de almacenarse en una pila en memoria.

Las próximas secciones permiten explorar motivaciones adicionales para el uso de las arquitecturas RISC y algunas características especiales que hacen que estas arquitecturas resulten efectivas.

10.3 Segmentación del trayecto de datos

El flujo de instrucciones a través de una unidad segmentada sigue los mismos pasos que cuando se ejecuta una instrucción. En el análisis siguiente se considerará la forma en que se ejecutan tres tipos de instrucciones: aritméticas, de salto y de acceso a memoria (carga y almacenamiento), y se relacionará esta ejecución con la forma en que se realiza la segmentación de las mismas.

10.3.1 Instrucciones aritméticas, de salto y de acceso a memoria

Si se considera la secuencia "normal" de eventos que se produce cuando se ejecuta una **instrucción aritmética** en una máquina con arquitectura de carga y almacenamiento, se tiene:

1. La búsqueda de la instrucción en memoria.
2. La decodificación de la instrucción (se trata de una instrucción aritmética, pero la CPU tiene que averiguarlo a través de la operación de decodificación).
3. La búsqueda de los operandos desde el conjunto de registros.
4. La transferencia de los operandos a la unidad aritmético-lógica.
5. El almacenamiento del resultado en el registro correspondiente.

Para distintas clases de instrucciones los patrones son similares. Para las **instrucciones de bifurcación**, la secuencia es la siguiente:

1. Búsqueda de la instrucción desde memoria.
2. Decodificación de la instrucción (es una instrucción de salto).
3. Búsqueda de los componentes de la dirección en la instrucción o en algún registro.
4. Transferencia de los componentes de la dirección a la unidad aritmética, para realizar el cálculo de la dirección.
5. Copia de la dirección resultante hacia el contador de programa, con lo que se realiza el salto.

La secuencia para una **instrucción de acceso a memoria,** ya sea de **carga** o de **almacenamiento**, es la siguiente:

1. Búsqueda de la instrucción en memoria.
2. Decodificación de la instrucción (es una instrucción de lectura o escritura en memoria).
3. Búsqueda de los componentes de la dirección a partir de la instrucción o del conjunto de registros.
4. Transferencia de los componentes de la dirección a la unidad aritmética, para realizar el cálculo de la dirección.
5. Utilización de la dirección efectiva obtenida para acceder a memoria junto con una señal de lectura o escritura. En caso de que se trate de una señal de escritura se debe recuperar del conjunto de registros el elemento de información a escribir.

Las tres secuencias muestran un alto grado de similitud en lo que se hace en cada etapa: (1) búsqueda, (2) decodificación, (3) búsqueda de operandos, (4) operación de la unidad aritmético-lógica, (5) escritura de los resultados.

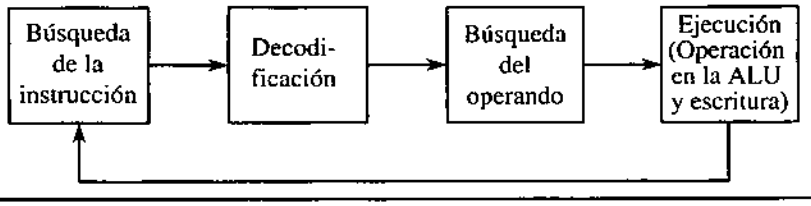


Figura 10.3 • Segmentación de instrucciones en cuatro etapas.

Estas cinco fases son similares a las cuatro fases analizadas en los capítulos 4 y 6, con la diferencia de haber dividido la fase de ejecución en dos subfases: la operación de la unidad aritmético-lógica y la escritura de los resultados, de acuerdo con lo que muestra la figura 10.3. No siempre es necesario volver a escribir el resultado, lo que se soluciona teniendo dos subfases separadas (ALU y reescritura) y un camino de alternativa para

saltear la escritura en aquellos casos en que no fuera necesaria. Para este análisis se ha optado por un planteo más simple, en el que todas las instrucciones deberán recorrer íntegramente cada fase, aun cuando no sea necesario.

10.3.2 Segmentación de las instrucciones

En la práctica, cada diseñador de una CPU determina el diseño del *pipeline** desde una perspectiva diferente, que dependerá de los objetivos particulares del diseño y del juego de instrucciones. Por ejemplo, la implementación original de SPARC incluye un *pipeline* de solo cuatro etapas de segmentación, en tanto que algunas unidades de punto flotante pueden tener una docena o más.

Durante el ciclo de búsqueda y ejecución, cada una de las unidades de ejecución realiza una tarea diferente. Una vez que la unidad de búsqueda de la instrucción completa su tarea, la instrucción recién obtenida se transfiere a la unidad de decodificación. En este momento, la unidad de búsqueda puede comenzar la búsqueda de la instrucción siguiente, la que se superpone con la ejecución de la instrucción anterior. Cuando las unidades de búsqueda y decodificación completan sus tareas, entregan las tareas restantes a las unidades siguientes (el próximo paso es la búsqueda de los operandos). El flujo de control continúa hasta que todas las unidades están ocupadas.

10.3.3 Cómo mantener ocupada la estructura

Debe observarse aquí un punto importante: si bien la ejecución de una instrucción en el modelo planteado lleva múltiples pasos, se puede ejecutar una instrucción por ciclo en la medida en que todas las etapas del *pipeline* estén ocupadas. No obstante, esto no ocurre, a menos que se ponga especial cuidado sobre la forma en que se ordenan las instrucciones. De la figura 10.1 se puede saber que aproximadamente una de cada cuatro instrucciones corresponde a un salto. No se puede realizar la búsqueda de una instrucción que viene después de un salto hasta que no se haya completado la ejecución del salto. Por lo tanto, cuando la estructura del *pipeline* está completa y se encuentra una instrucción de salto, debe **limpiarse** la estructura, lo que se hace completándola con instrucciones de no operar (NOP). Algo similar ocurre con las instrucciones de LOAD y STORE. Estas instrucciones generalmente requieren un ciclo de reloj adicional en el que acceden a memoria, lo que produce el efecto de expandir la fase de ejecución a dos ciclos de reloj. Los ciclos de espera se completan con instrucciones NOP.

La figura 10.4 ilustra el comportamiento del *pipeline* correspondiente al juego de instrucciones de ARC durante una referencia a memoria y durante una instrucción de salto. La instrucción addcc ingresa en la estructura en el instante de tiempo (ciclo) 1. En el ciclo 2, se in-

* *N. de T.*: Se ha optado por seguir utilizando la palabra *pipeline*, tal como se la usa en el original, para definir la estructura de la unidad de control dividida en etapas con diferentes funciones, aún cuando en otras secciones del libro se ha utilizado la referencia a unidades segmentadas.

gresa la instrucción `ld`, que hace referencia a memoria, en tanto que `addcc` pasa a la etapa de decodificación. El *pipeline* continúa completándose con las instrucciones `srl` y `subcc` en los ciclos 3 y 4, respectivamente. En el ciclo 4 se ejecuta la instrucción `addcc`, la que, por consiguiente, sale de la estructura. En el ciclo 5 la instrucción `ld` alcanza la etapa de ejecución, pero no la finaliza debido a la necesidad de un ciclo adicional de reloj para la referencia a memoria. La instrucción `ld` finaliza su ejecución durante el ciclo 6.

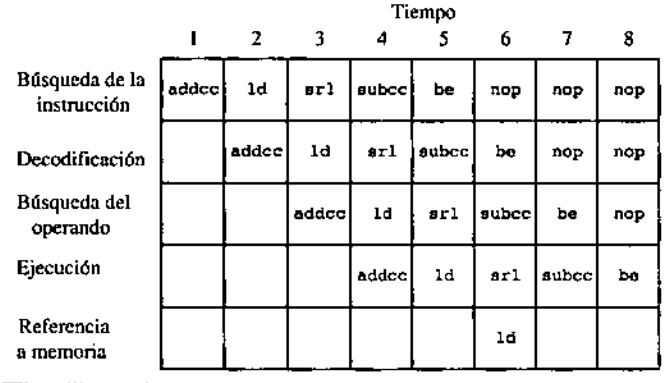


Figura 10.4 • Conducta de un *pipeline* durante una referencia a memoria y durante un salto.

Retardos de carga y de salto

Tanto la instrucción de carga, `ld`, como la de almacenamiento, `st`, requieren cinco ciclos de reloj, en tanto que las restantes instrucciones requieren solo cuatro. Por lo tanto, la instrucción que sigue a una instrucción de carga o de almacenamiento no debería utilizar el registro que estas instrucciones están cargando o almacenando. Tal como se muestra en la figura 10.5a, una buena solución consiste en incluir una instrucción `NOP` luego de una instrucción `ld` o `st`. El ciclo (o ciclos, dependiendo de la arquitectura) adicional requerido para la carga se conoce como de **carga retardada**, debido a que la información cargada no se encuentra disponible inmediatamente en el ciclo siguiente al de la carga. El caso del **salto retardado** es similar, tal como se puede observar en la secuencia correspondiente a la instrucción `be`, entre los ciclos 5 y 8 de la figura 10.4. La posición ocupada por la instrucción `NOP` suele denominarse **ranura de retardo de carga** o **ranura de retardo de salto**, respectivamente.

En muchos casos, el compilador puede llegar a encontrar instrucciones capaces de completar las ranuras de retardo. En la figura 10.5a, la instrucción `srl` puede tomar la posición de la `nop` debido a que su utilización de registros no entra en conflicto con los códigos que lo rodean. Por lo tanto, reordenar las instrucciones en esta forma no influye sobre el resultado. Luego de reemplazar la instrucción `nop` por la instrucción `srl`, se obtiene la codificación de la figura 10.5b. Este esquema es el que se ha representado en la figura 10.4 durante su pasaje a través del *pipeline*.

```
srl    %r3, %r5
addcc  %r1, 10, %r1
ld     %r1, %r2
nop
subcc  %r2, %r4, %r4
be     label
```

(a)

```
addcc  %r1, 10, %r1
ld     %r1, %r2
srl    %r3, %r5
subcc  %r2, %r4, %r4
be     label
```

(b)

Figura 10.5 • Código SPARC (a) con el agregado de una instrucción `nop`; (b) con la instrucción `srl` desplazada a la posición del `nop`.

Ejecución especulativa de las instrucciones

Un enfoque alternativo que permite manejar el comportamiento de los saltos consiste simplemente en adivinar cuál será el camino por el que siga el programa tras el salto, y luego deshacer cualquier daño que se haya producido por haber tomado el camino equivocado. Estadísticamente, en el caso de un lazo cerrado en un programa, son más las veces en que se lo ejecuta que las que no se lo ejecuta, por lo que suele ser un buen intento el de suponer que no se va a tomar el camino de bifurcación hacia la salida del lazo. Por lo tanto, el procesador puede comenzar a procesar la instrucción siguiente adelantándose al resultado del salto. Si el salto deriva hacia el camino equivocado (el que no ha sido elegido), puede detenerse la fase de ejecución de la instrucción siguiente, así como cualquier instrucción subsiguiente que haya ingresado en el *pipeline*, con el objeto de limpiar la estructura. Esta solución funciona adecuadamente en una buena cantidad de arquitecturas, particularmente en aquellas que tienen ciclos de reloj lentos o estructuras con muchos niveles de segmentación. No obstante, en las arquitecturas RISC, suele ser muy grande el tiempo que se pierde en la determinación del momento en que el salto va a salir por el lado equivocado y en la limpieza de los efectos secundarios producidos por la carga de instrucciones equivocadas. Las arquitecturas RISC, normalmente, utilizan instrucciones `NOP` cuando no encuentran nada útil con que reemplazarlas.

Ejemplo: análisis de la eficiencia de una estructura segmentada

En este ejemplo se analiza la eficiencia de una estructura segmentada. Se supone un procesador con un *pipeline* de cinco etapas. Si se debe ejecutar un salto, se requieren cuatro ciclos para limpiar la estructura. Por lo tanto, la penalización por el salto vale $b = 4$. La probabilidad de que una instrucción determinada sea un salto es de $P_b = 0{,}25$. La probabilidad de que se realice el salto es de $P_t = 0{,}5$. Se desea calcular el número promedio de ciclos requeridos para la ejecución de una instrucción y la **eficiencia de ejecución**.

Cuando la estructura está completa y no hay saltos, el número promedio de ciclos por instrucciones (CPI_{nob}) es 1. Cuando existen saltos, la cantidad promedio de ciclos por instrucción es de:

$$CPI_{medio} = (1 - P_b)(CPI_{nob}) + P_b[P_t(1 + b) + (1 - P_t)(CPI_{nob})$$
$$= 1 + P_b P_t$$

Luego de los reemplazos correspondientes, se tiene:

$$CPI_{medio} = (1 - 0{,}25)(1) + 0{,}25[0{,}5(1 + 4) + (1 - 0{,}5)(1)] = 1{,}5 \text{ ciclos}$$

La eficiencia de ejecución puede definirse como la relación entre la cantidad de ciclos por instrucción cuando no hay saltos y la misma cantidad cuando los hay. En este caso resulta:

$$\text{Eficiencia de ejecución} = (CPI_{nob})/(CPI_{medio}) = 1/1{,}5 = 67\%$$

Como consecuencia de las instrucciones de salto, el procesador funciona al 67% de su velocidad potencial, pero esto todavía es mucho mejor que los cinco ciclos por instrucción que se requerirían sin segmentación.

Existen técnicas para la mejora de la eficiencia. Como se ha dicho antes, se sabe que los lazos cerrados se ejecutan habitualmente más de una vez; en consecuencia, si se define que el camino de un salto que lo saca del lazo no es el camino que el programa va a seguir, se estará en lo cierto en la mayoría de los casos. También pueden ejecutarse programas de simulación sobre las instrucciones de salto que no corresponden a lazos, y obtener una muestra estadística de cuáles son los saltos que probablemente sí se ejecuten, tras lo cual se podrá anticipar el comportamiento en forma coherente. Tal como se ha dicho con anterioridad, estas técnicas funcionan bien cuando el *pipeline* tiene muchas etapas o el reloj es lento. •

10.4 Superposición de ventanas de registros

La **superposición de ventanas de registros** es una prestación arquitectónica moderna que no ha sido tan ampliamente adoptada como otras, dado que en la actualidad solo la utiliza la arquitectura SPARC. Esta característica se basa en estudios que muestran que los programas habitualmente pierden gran cantidad de tiempo manejando llamadas a procedimientos y sus retornos, lo que implica la transferencia de parámetros a través de una pila, que en las arquitecturas tradicionales se encuentra ubicada en la memoria principal. La arquitectura SPARC disminuye apreciablemente la pérdida de tiempo involucrada, por medio del uso de múltiples conjuntos de registros, los que se superponen. Estos registros se utilizan para realizar la transferencia de parámetros entre procedimientos, en lugar de utilizar una pila ubicada en la memoria del sistema.

En un programa convencional puede existir un encadenamiento profundo de los procedimientos, pero si se analiza una ventana de tiempo determinada, esa profundidad de encadenamiento fluctúa en una banda angosta. La figura 10.6 ilustra este comportamiento. Si se considera una ventana de profundidad de encadenamiento con un tamaño de 5 niveles, la ventana se mueve solo 18 veces en 100 llamados a procedimiento. Los resultados de análisis realizados por un grupo de la Universidad de California en Berkeley (véase Y. Tamir y C. Sequin) muestran que si se toma un tamaño de ventana de 8 niveles solo se producirán variaciones en menos del 1% de los llamados o retornos.

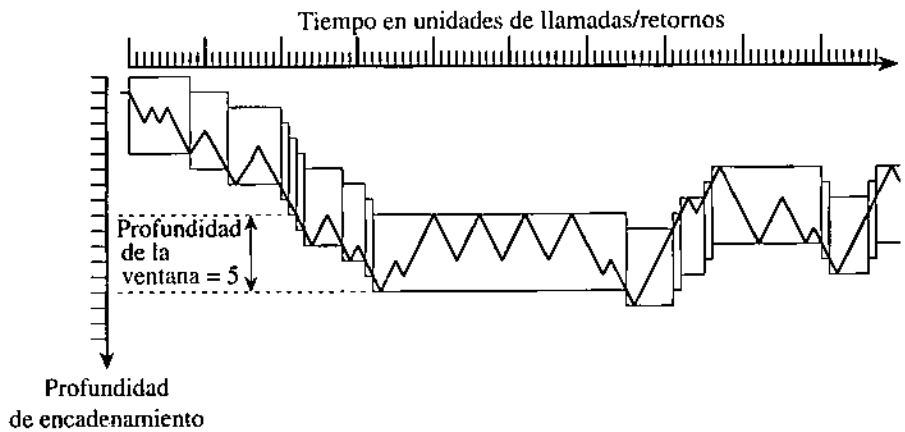


Figura 10.6 • Comportamiento de llamadas y retornos en función de la profundidad de encadenamiento y del tiempo (adaptado de W. Stallings).

Para mejorar el rendimiento es necesario que el tamaño de las ventanas para llamadas encadenadas sea pequeño. Normalmente, para cada llamado a procedimiento se construye una zona en la pila, en la que se colocan los parámetros, la dirección de retorno y las variables locales. Cuando se realiza un llamado a una subrutina o procedimiento, se requiere un gran trabajo de manejo de la pila, pero la complejidad del manejo no es tan grande, dado que las referencias a la pila se encuentran muy localizadas en un área pequeña.

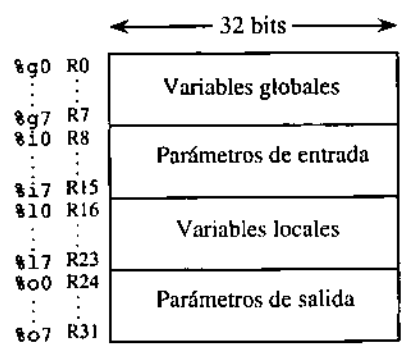


Figura 10.7 • Visión del usuario de los registros de RISC I.

La arquitectura RISC I aprovecha esta localización, por lo que mantiene la sección activa de la pila en un conjunto de registros. La figura 10.7 ilustra la imagen que puede ver un usuario en cuanto al empleo de los registros en RISC I. El usuario ve 32 registros, cada uno de ellos de 32 bits de palabra. Los registros R0-R7 se utilizan para las variables globales. Los registros R8-R15 se utilizan para los parámetros de entrada. Los registros R16-R23 se usan para las variables locales, en tanto que los registros R24-R31 se utilizan para los parámetros de salida. Los ocho registros de cada grupo son suficientes para satisfacer la gran mayoría de los requerimientos de la actividad de llamada y retorno, según lo muestra la distribución de la figura 10.6.

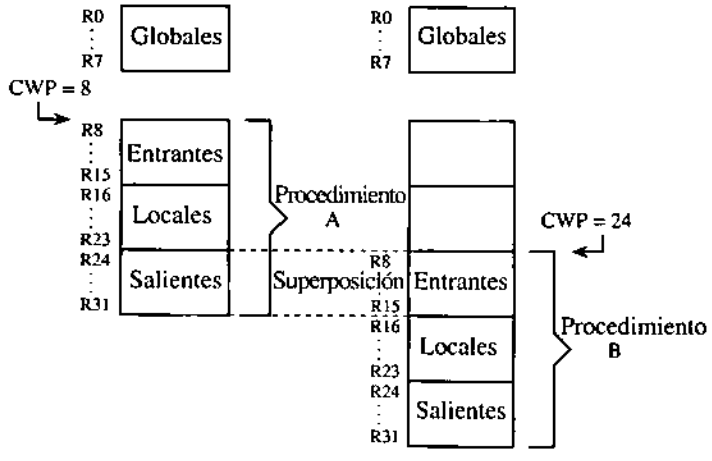


Figura 10.8 • Superposición de ventanas de registros.

Si bien el usuario ve 32 registros, puede haber varios cientos de registros que se superpongan. La figura 10.8 ilustra el concepto de ventanas de registros superpuestas. Los registros globales se separan de los restantes y se tienen disponibles en forma continua como R0-R7. Los registros R8-R31 forman un único bloque de 24 registros visibles para el usuario, aun cuando en cada llamado a procedimiento este grupo de registros se desliza hacia la profundidad del **archivo de registros** (el conjunto completo de registros de la arquitectura). Dado que los parámetros de salida de un procedimiento son los parámetros de entrada a otro, estos conjuntos de registros pueden superponerse. El conjunto de registros R8-R31 se conoce como una ventana. El **puntero a la ventana actual** (CWP, *current window pointer*) que apunta a la ventana siguiente se incrementa o decrementa, respectivamente, en cada llamada o retorno.

Solo en aquellas situaciones estadísticamente raras en que la cantidad de registros no sea suficiente para la cantidad de anidamientos producida, se utiliza memoria. No obstante, la memoria principal se usa para la ventana de menor numeración, de modo que la nueva ventana actual sigue utilizando registros. El bloque de registros superior se cierra sobre el inferior, con lo que se forma un ***buffer* circular**. A medida que se van produciendo los retornos, los registros que habían sido volcados a la memoria se reintegran al ar-

chivo de registros. Por lo tanto, la ejecución siempre se realiza asociada con registros y nunca en forma directa con la memoria principal.

Ejemplo: código compilado para ventanas superpuestas y saltos retardados

En esta sección se analiza un programa escrito en lenguaje simbólico SPARC, producido por la salida de un compilador C, que aprovecha algunos de los conceptos y características de la concepción RISC: el programa en lenguaje C se presenta en la figura 10.9. Se trata de un programa cuya rutina principal transfiere dos valores enteros a una subrutina, la que devuelve a la rutina principal la suma de dichos enteros.

```
/* Ejemplo de programa C a compilar con gcc */

#include
<stdio.h>
main ()
{
    int a, b, c;

    a = 10;
    b = 4;
    c = add_two(a, b);

    printf("c = %d\n", c);
}

int add_two(a,b)
int a, b;
{
    int result;

    result = a + b;
    return(result);
}
```

Figura 10.9 • Código fuente en lenguaje C de un programa a ser compilado con gcc.

La figura 10.10 muestra el código producido por un compilador C Solaris, usando la línea de comandos

```
gcc-S file.c
```

La figura 10.10 explica los aspectos más significativos del código ensamblado, el que tiene varias propiedades que solo se encuentran en el código RISC. Se han incluido en el código una cantidad de nuevas instrucciones y directivas:

`.seg/.section`. Los programas ejecutables en UNIX tienen tres segmentos para datos, texto (las instrucciones) y la pila. La directiva `.seg` le indica al ensamblador que coloque el código siguiente en uno de esos tres segmentos. Algunos de los segmentos tienen

```
! Salida generada por el compilador Solaris (UNIX Sun)
! Comentarios agregados por el autor

.file   add.c   ! Identifica al programa principal
.section        ".rodata"       ! Datos de lectura para la rutina main
        .align 8                ! Alinea los datos de lectura para la rutina
                                ! main en dirección múltiplo de 8
.LLC0
        .asciz  "c = %d\n"      ! Estos son los datos de lectura
.section        "text"  ! Aquí comienza el código ejecutable
        .align 4  ! Alinea el código ejecutable en dirección múltiplo de 4 bytes
        .global main
        .type   main,#function
        .proc   04
main:           ! Comienzo del código ejecutable correspondiente a la rutina main
        !#PROLOGUE#     0
        save %sp, -128, %sp  ! Crear en la pila una zona de datos de 128 bytes
                             ! Avanzar el CWP (puntero a ventana actual)
        !#PROLOGUE#     1
        mov 10, %o0  ! %o0 <- 10. Nótese que el registro %o0 equivale al %r4.
    ! Esta es la variable local a de la rutina principal del programa fuente en C.
        st %o0, [%fp-20]      ! Almacenar el registro %o0 cinco palabras dentro de la zona de datos.
        mov 4, %o0   ! %o0 <- 4. Esta es la variable b de la rutina main.
        st %o0, [%fp-24]      ! Salvar el registro %o0 seis palabras dentro de la zona de datos.
        ld [%fp-20], %o0      ! Cargar %o0 con los parámetros a transferir
        ld [%fp-24], %o1      ! a la rutina add_two.
        call add_two, 0       ! Llamado a la rutina add_two
        nop     ! Luego de transferir el control se requiere la limpieza de pipeline
        st %o0, [%fp-28]      ! Almacenar el resultado 67 bytes dentro de la zona de datos.
                              ! Esta es la variable local c de main.
        sethi %hi(.LLC0), %o1 ! Esta instrucción y la siguiente cargan en %o0 la dirección
        or %o1, %lo(.LLC0), %o0 ! .LLC0 de 32 bits
        ld [%fp-28], %o1  ! Cargar %o1 con el parámetro a tranferir a printf
        call printf, 0
        nop     ! Se requiere un nop por la limpieza del pipeline que
                ! sigue a una transferencia de control.
.LL1
        ret     ! Retorno a la rutina invocante (Solaris, en este caso)
        restore ! El complemento a salvar. Si bien está luego de la
            ! instrucción de retorno, sigue en el pipeline y se ejecuta.
.LLfe1
        .size   main, .LLfe1-main ! Tamaño
        .align 4
        .global add_two
        .type   add_two, #function
        .proc 04
add_two:
         !#PROLOGUE# 0
         save %sp, -120, %sp
         !#PROLOGUE# 1
         st %i0, [%fp+68] !Igual a %o0 en la rutina invocante (variable a)
         st %i1, [%fp+72] !Igual a %o1 en la rutina invocante (variable b)
         ld [%fp+68], %o0
         ld [%fp+72], %o1
         add %o0, %o1, %o0 ! Realizar la suma
         st %o0, [%fp-20]  ! Almacenar resultado en zona de datos
         ld [%fp-20], %i0  ! %i0 (resultado) es %o0 en la rutina invocante
         b .LL2
         nop
.LL2:
         ret
         restore
.LLfe2:
         .size   add_two, .LLfe2-add_two
         .ident  "GCC: (GNU) 2.5.8"
```

Figura 10.10 • Código SPARC generado por gcc.

protecciones diferentes, lo que justifica la razón de la existencia de un segmento data y de un segmento datal. El segmento datal contiene constantes y, por lo tanto, debe estar protegido contra escritura. El segmento data es de lectura y escritura y, por consiguiente, no está protegido contra lectura ni escritura (aun cuando, al igual que el segmento datal, está protegido para que no sea ejecutado). Las versiones más modernas de UNIX permiten definir más áreas de texto y datos, con distintas protecciones contra lectura, escritura y ejecución.

`%hi` es la directiva ARC `.high22`.

`%lo` es la directiva ARC `.low10`.

`add` es una instrucción de suma (como `addcc`) que no afecta a los códigos de condición.

`save` avanza el puntero a la ventana actual e incrementa el puntero a la pila para crear espacio para las variables locales.

`mov` equivale a `or %g0, registro_o_inmediato,registro_destino`. Difiere de la instrucción de almacenamiento `st` en que el destino es un registro.

`nop`: no operar (el procesador espera un ciclo de instrucción mientras se completa una instrucción de salto).

`.ascii/.asciz` reserva espacio para un conjunto de caracteres ASCII.

`set` fija un registro en un valor dado. Es una macroinstrucción que se expande a las construcciones `sethi`, `%hi` y `%lo` del `#PROLOGUE# 1`.

`ret`: retorno. Equivale a `jmpl %i7+8, %g0`. Si bien parecería que la instrucción de retorno debe volver a una posición desplazada 4 bytes de la instrucción de llamada, el retorno se hace 8 bytes más allá debido a que la instrucción que sigue a la instrucción de llamada es un `nop`, insertado por el compilador para proteger la integridad del *pipeline*.

`restore` decrementa el puntero a la ventana actual.

`b` equivale a `ba`.

`.file` identifica el archivo fuente.

`.align` fuerza al código que le sigue para que se ubique en una frontera exactamente divisible por su argumento.

`.type` asocia un rótulo con su tipo.

`.size` calcula el tamaño de un segmento.

`.ident` identifica la versión del compilador.

Se pueden ver las posiciones de las ranuras de retardo, marcadas con instrucciones `nop`. (No se ha utilizado todavía la característica optimizadora del compilador.) A pesar de la disponibilidad de ventanas superpuestas de la arquitectura SPARC, el código no optimizado no utiliza esta prestación. Los parámetros a ser transferidos a una rutina se copian sobre la pila, y la rutina invocada rescata estos parámetros de la pila. Otra vez, no se ha tomado en cuenta aún la prestación de optimización del compilador (pero conviene seguir leyendo).

Nótese que el compilador no parece ser muy coherente con su elección de registros para la transferencia de parámetros. Antes del llamado a `add_two`, el compilador usa `%o0` y `%o1` (`%r0` y `%r1`) para los parámetros que se transfieren a dicha rutina. Luego, se usa `%r1` para el pasaje de parámetros a `printf`. ¿Cuál es la causa por la que el compilador no vuelve a usar `%r0`, o por qué no elige el registro disponible siguiente (`%o2`)? El **problema de la asignación de registros** ha sido objeto de gran cantidad de estudios. No se pretende entrar en detalles en este texto[1] dado que el tema es más adecuado para un curso de diseño de compiladores, pero alcanza con decir que cualquier asignación correcta de variables a registros funciona, aunque algunas asignaciones son mejores que otras en cuanto a la cantidad de registros usados y al tiempo total de ejecución del programa.

¿Por qué son tan grandes los espacios requeridos en la pila? Solo se requieren tres palabras en la pila para contener las variables `a`, `b` y `c` en `main`. Puede hacer falta una palabra adicional para almacenar la dirección de retorno, si bien no parece que el compilador generase código con ese objeto. El sistema operativo no le transfiere parámetros a la rutina `main`, por lo que el espacio de memoria que `main` requiere ver es de solo cuatro palabras (16 bytes). Por lo tanto, la línea con la que comienza la rutina `main`:

```
save %sp, -128, %sp
```

debería ser simplemente:

```
save %sp, -16, %sp
```

¿Para qué hace falta todo el resto del espacio? Existe una cantidad de situaciones que pueden requerir espacio en la pila durante el tiempo de ejecución. Por ejemplo, si la profundidad del anidamiento es mayor que la cantidad de ventanas, se debe utilizar la pila para controlar el desborde (véase la figura D.2 en el texto de SPARC).

Si una rutina transfiere un escalar a otra, todo funciona bien. Pero si quien invoca hace referencia a la dirección de un escalar (o agregado) transferido, el mismo debe copiarse en la pila y luego debe ser invocado desde la pila durante toda vida del puntero (o la vida del procedimiento, si no se conoce la del puntero).

¿Cuál es la razón por la que la sentencia `ret` de retorno vuelve al código que se encuentra 8 bytes después del `call`, en lugar de 4 bytes, como se ha venido pregonando? Ya se ha dicho previamente que esto se debe a la existencia de un `nop` posterior al `call` (una instrucción de retardo).

Nótese que los rótulos de rutinas que aparecen en el código fuente contienen un prefijo consistente en un guión bajo, en el código simbólico, de modo tal que `main`, `add_two` y

1. Para los curiosos, aquí hay algunos detalles: %r0 (%o0) sigue en uso (`add_two` espera que en %r0 aparezca la dirección de `LLC0`), y `%r1` ya no hace falta en este punto, por lo que se lo puede reasignar. Pero entonces, ¿por qué se utiliza `%r1` en la línea `sethi`? ¿No hubiese tenido sentido usar `%r0` en lugar de hacer aparecer otro registro en el cálculo? El problema 10.2, al final del capítulo, agrega más sobre el tema.

`printf` del programa C se convierten en `_main`, `_add_two` y `_printf` en el código SPARC generado por `gcc`. Esto significa que si se desea escribir un programa en C que se enlace con un programa SPARC generado por `gcc`, las llamadas del programa de C deben hacerse a rutinas cuyo nombre comience con el guión bajo. Por ejemplo, si se compila la rutina `add_two` dentro de un código SPARC, y se la invoca desde otro programa escrito en C en otro archivo, el programa escrito en C deberá invocar a la rutina `_add_two` y no a la rutina `add_two`, aun a pesar de haber nacido la rutina como `add_two`. Más aún, el programa escrito en C debe declarar a la rutina simbólica `_add_two` como externa.

Si se continúa con la compilación de `add_two` hasta el programa ejecutable, no hay necesidad de tratar a los rótulos en forma diferente. La rutina `add_two` seguirá teniendo el rótulo `_add_two`, pero la rutina `main` se compilará en un código que espera encontrar `_add_two`, con lo que todo funcionará en la forma esperada. Sin embargo, no es este el caso si el programa `gcc` invoca programas de una biblioteca Fortran.

Todavía, Fortran es un lenguaje de uso habitual en la comunidad científica, por lo que existe una cantidad de bibliotecas Fortran que se utilizan en álgebra lineal, modelado y simulación, y en aplicaciones científicas paralelas. Quienes programan en un lenguaje XYZ (cualquiera sea), suelen encontrarse con la necesidad de escribir programas en XYZ que hagan uso de rutinas Fortran. Esto es simple una vez que se entiende lo que está ocurriendo.

Existen dos hechos significativos que deben tenerse en cuenta:

1. Diferencias en los rótulos de las rutinas.
2. Diferencias en el enlace de las subrutinas.

En Fortran, los rótulos del código fuente requieren un prefijo formado por dos guiones bajos en el código ensamblador. Un programa escrito en C (si fuese C el lenguaje XYZ) que haga un llamado a la rutina Fortran `add_two` debería invocar la rutina `_add_two`, la que también debe ser declarada como externa en el código fuente C (y declarada como global en el programa Fortran).

Si todos los parámetros que se transfieren a las rutinas Fortran son punteros, todo funciona normalmente. Si se realiza una transferencia de escalares, habrá algún tipo de problemas debido a que C (tal como Java) utiliza llamados a escalares por valor, en tanto que Fortran los invoca por referencia. Se requiere "engañar" al compilador C para que utilice llamado por referencia, haciéndolo explícito. Cada vez que una rutina Fortran espere un escalar en su línea de argumentos, en el código C se utilizará un puntero a dicho escalar.

Como consideración práctica, el compilador `gcc` puede compilar programas Fortran. Por medio de la observación de la extensión del archivo fuente, que debe ser `.f` para rutinas Fortran, el compilador sabe qué es lo que debe hacer.

Ahora corresponde ver cuál es la forma en que un compilador optimizador puede mejorar el código. La figura 10.11 ilustra el código optimizado obtenido por medio del uso de la bandera `–O` del compilador. Nótese que no hay ni una sola instrucción `nop`, `ld` o `st`. Se han recuperado los ciclos previamente derrochados mediante el uso de instrucciones `nop` y se han eliminado además las referencias a memoria dedicadas a manejos de la pila.

```
! Salida producida por el compilador gcc cuando se
utiliza la variable de optimización -O del compilador

.file   "add.c"
.section        ".rodata"
        .align 8
.LLC0:
        .asciz  "c = %d\n"
.section        ".text"
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:
        !#PROLOGUE# 0
        save %sp,-112,%sp
        !#PROLOGUE# 1
        mov 10,%o0
        call add_two,0
        mov 4,%o1
        mov %o0,%o1
        sethi %hi(.LLC0),%o0
        call printf,0
        or %o0,%lo(.LLC0),%o0
        ret
        restore
.LLfe1:
        .size    main,.LLfe1-main
        .align 4
        .global add_two
        .type    add_two,#function
        .proc   04
add_two:
        !#PROLOGUE# 0
        !#PROLOGUE# 1
        retl
        add %o0,%o1,%o0
.LLfe2:
        .size    add_two,.LLfe2-add_two
        .ident  "GCC: (GNU) 2.7.2"
```

Figura 10.11 • Código SPARC generado con la bandera de optimización –O.

10.5 Máquinas con instrucciones múltiples (superescalares). El PowerPC 601

En los análisis realizados anteriormente con relación a la segmentación, se puede observar que en un mismo momento se tienen distintas instrucciones en distintas etapas de su ejecución. Aquí se analiza la arquitectura superescalar, en la cual se pueden ejecutar varias instrucciones en forma simultánea haciendo uso de unidades de ejecución separadas. En una arquitectura superescalar pueden existir una o más **unidades de enteros** (IU, *integer units),* **unidades de punto flotante** (FPU, *floating point units*) y de **procesamiento de saltos** (BPU, *branch processing units*). Esto implica que las instrucciones se deben asignar a las diversas unidades de ejecución y, más aún, que las instrucciones pueden ejecutarse fuera de secuencia.

La ejecución fuera de secuencia implica que las instrucciones deben ser examinadas antes de ser **derivadas** a una unidad de ejecución, no solo para determinar a cuál de las unidades de ejecución corresponde su ejecución, sino para determinar también si la ejecución desordenada de las instrucciones puede provocar una falla en la ejecución total del programa debido a las dependencias existentes entre las instrucciones. A su vez, esto requiere la existencia de una **unidad de instrucciones** que pueda realizar una búsqueda an-

ticipada de las instrucciones colocándolas en una cola de instrucciones, determinar los tipos de instrucciones y las relaciones de dependencia entre las mismas, y distribuirlas entre las distintas unidades de ejecución.

10.6 Estudio de un caso: el procesador PowerPC como arquitectura superescalar

Como ejemplo de arquitectura superescalar moderna se procederá a examinar el procesador Motorola PowerPC® 601. En la actualidad, el 601 ha sido superado por miembros más potentes de la familia PowerPC, pero servirá igualmente para ilustrar las características importantes de las arquitecturas superescalares sin introducir en el análisis ninguna complejidad innecesaria.

10.6.1 La arquitectura de programación del PowerPC 601

El 601 es un procesador RISC con 32 registros de uso general cuya arquitectura de programación incluye:

- Treinta y dos registros de uso general para enteros (GPR), de 32 bits.
- Treinta y dos registros de punto flotante (FPR) de 64 bits.
- Ocho registros de códigos de condición, de 4 bits.
- Alrededor de 50 registros dedicados, de 32 bits, que se utilizan para el control de distintos aspectos de la administración de memoria y del sistema operativo.
- Más de 250 instrucciones, muchas de las cuales son de propósitos especiales.

10.6.2 La arquitectura de hardware del PowerPC 601

La figura 10.12, adaptada del manual de usuario del PowerPC 601, ilustra la microarquitectura del 601. El flujo de instrucciones y de datos se desplaza a través de la interfaz del sistema, que se ubica en la parte inferior de la figura, hacia la memoria *cache* de 32 Kbytes. Desde ahí, las instrucciones se cargan, de a ocho por vez, en la unidad de instrucciones, ubicada en la parte superior de la figura. La lógica de análisis incluida en la unidad de instrucciones examina las instrucciones de la cola para verificar sus tipos y dependencias, y las deriva hacia una de las tres unidades de ejecución IU, BPU o FPU.

En la figura se indica que la unidad de enteros (IU) contiene el conjunto de registros de uso general y un registro de excepción de enteros (XER), que contiene información referida a cualquier excepción que pudiese producirse dentro de la unidad. La unidad de enteros puede ejecutar la mayor parte de las instrucciones referidas a enteros en un solo ciclo de reloj, lo que evita la necesidad de segmentación alguna en este tipo de instrucciones.

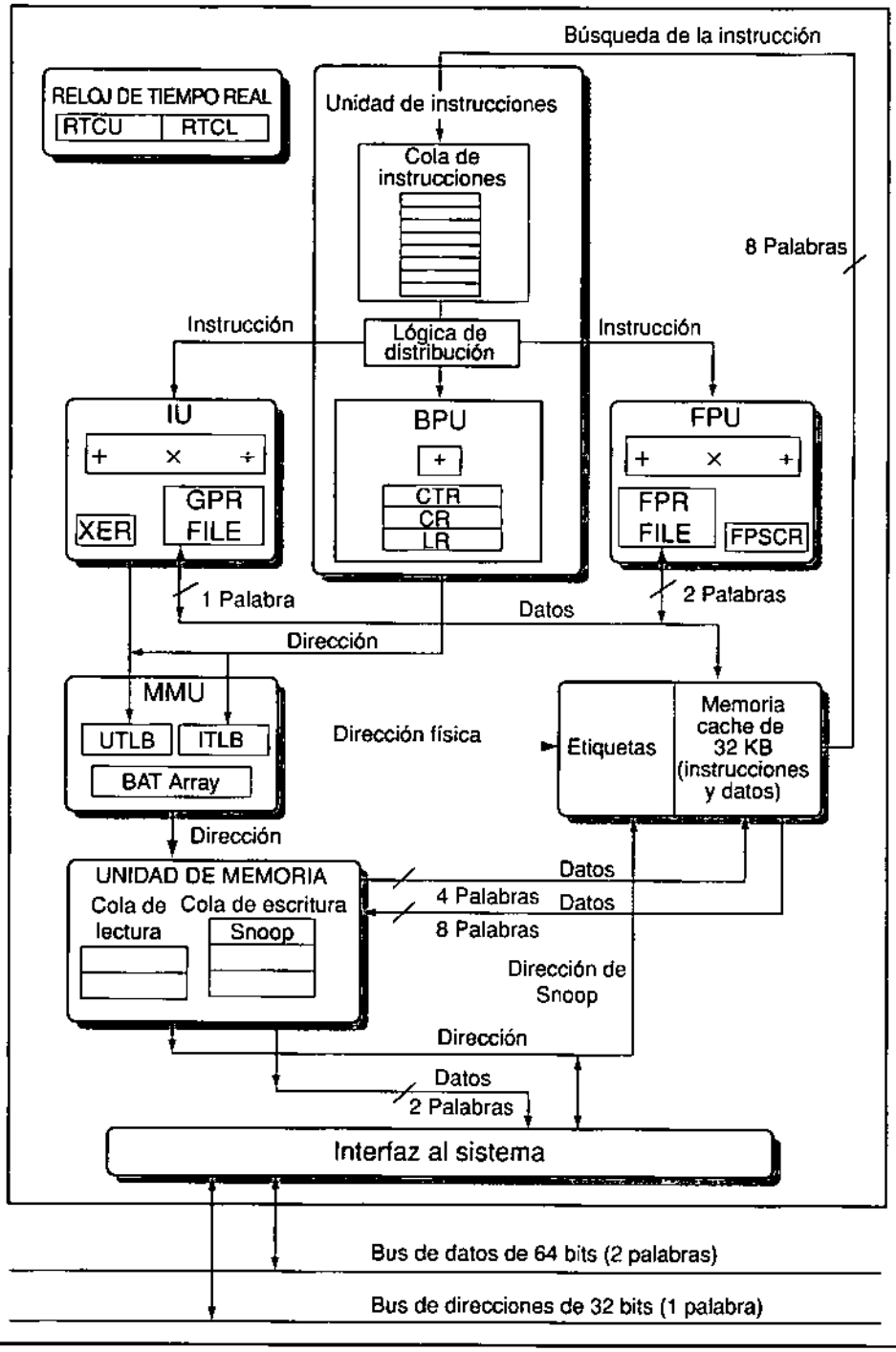


Figura 10.12 • La arquitectura del PowerPC 601. Adaptado del manual de usuario del procesador Motorola PowerPC 601.

La unidad de punto flotante FPU contiene los registros de punto flotante y el registro de control y estado de punto flotante (FPSCR). Este registro tiene información acerca de las excepciones de punto flotante y del tipo de resultados producidos por la unidad de punto flotante. La unidad de punto flotante está segmentada, y la mayoría de las instrucciones de punto flotante pueden atenderse a razón de una por ciclo de reloj.

Como se ha mencionado antes en la sección referida a segmentación, las instrucciones de salto, especialmente las de salto condicional, generan un cuello de botella cuando se trata de superponer la ejecución de las instrucciones. Esto se debe a que, previamente, hay que verificar que la condición del salto sea cierta (por ejemplo, un salto por positivo debe verificar que la bandera N que indica un resultado negativo esté en su estado falso). Recién entonces puede calcularse la dirección del salto, lo que muchas veces implica cálculo aritmético para determinar la dirección. Solo luego de este momento puede cargarse el contador de programa con la dirección hacia la que se debe saltar.

El 601 ataca este problema en distintas formas. Primero, de acuerdo con lo ya mencionado, existen ocho registros de códigos de condición, de cuatro bits cada uno, en vez de un único registro como es tradicional. Esta característica da por resultado registros de estado independientes para un conjunto de hasta ocho instrucciones distintas, por lo que ninguna de ellas puede interferir con ninguna otra en el manejo de los bits de códigos de condición. La unidad de procesamiento de saltos analiza la cola de instrucciones y, si detecta una instrucción de salto condicional, procede a calcular anticipadamente la dirección de destino; además, busca las instrucciones correspondientes a dicha dirección de destino. Si se ejecuta, el resultado es el de un salto efectivamente realizado en cero ciclos, dado que la instrucción correspondiente a la dirección de salto ya ha sido obtenida con anticipación a que se verificara el cumplimiento de la condición del salto. La unidad de procesamiento de saltos incluye, además, un registro de enlace, en el que puede almacenar direcciones de retorno de subrutinas, por lo que permite ahorrar un registro de uso general así como varios otros registros utilizados para aplicaciones especiales. Nótese que, con el objeto de anticipar la búsqueda de las instrucciones, la unidad de procesamiento de saltos puede enviar sus direcciones hacia las unidades de memoria y de administración de memoria por un bus separado.

La unidad que en la figura se denomina RTC consiste en un reloj de tiempo real, que posee un calendario con un rango de 137 años y con una precisión de 1 ns.

Las unidades de memoria y de administración de memoria (MMU) colaboran en la búsqueda tanto de las instrucciones como de los datos. Nótese el recorrido especial para datos que va directamente desde la memoria *cache* hacia los conjuntos de registros de punto fijo (enteros) y de punto flotante.

El procesador PowerPC 601 y sus descendientes más poderosos representan arquitecturas típicas del diseño moderno de microprocesadores de uso general. Las actuales familias de microprocesadores son de diseño superescalar, y habitualmente tienen varias unidades de cada tipo de unidad de ejecución.

10.7 Máquinas con palabra muy larga de instrucción

La arquitectura que de alguna manera compite con la arquitectura superescalar es la que se conoce como arquitectura **VLIW** (*very large instruction word*, palabra muy larga de instrucción). En las máquinas VLIW se empaquetan múltiples operaciones en una única palabra de instrucción que puede tener 128 o más bits. Las máquinas VLIW tienen múltiples unidades de ejecución, en forma similar a las estructuras superescalares. Una CPU VLIW típica podrá tener dos unidades de enteros, dos de punto flotante, dos unidades de carga y almacenamiento, y una unidad de procesamiento de saltos. La organización de las distintas operaciones que forman la palabra múltiple es responsabilidad del compilador. Esto libera a la CPU de tener que examinar las instrucciones para verificar su dependencia, o de tener que ordenarlas o reordenarlas. Como desventaja, el compilador debe ser pesimista en su análisis de dependencias. Si no puede encontrar la cantidad suficiente de instrucciones como para completar la palabra de instrucción, debe llenar los agujeros vacíos con instrucciones de no operar (NOP). Más aún, las mejoras de las arquitecturas VLIW requieren que se recompile el software para hacer uso de las mismas.

Se han hecho varios intentos de comercialización de maquinas VLIW, pero, en su mayoría, estas maquinas no han tenido una buena recepción en los años recientes. Su rendimiento es su principal objeción, por las razones expuestas entre otras.

10.8 Estudio de un caso: la arquitectura Intel IA-64 (Merced)

Esta sección analiza una familia de microprocesadores que se encuentra en desarrollo por parte de una alianza entre Intel y Hewlett Packard, de la que se espera haga entrar al consorcio en el siglo XXI. Se analizan, primeramente, los antecedentes que condujeron a la decisión de desarrollar una arquitectura nueva, tras lo cual se expone lo que actualmente se conoce de esa nueva arquitectura. (La información que se incluye en esta sección ha sido tomada de diversas publicaciones y sitios web, no habiendo sido confirmada por Intel ni por Hewlett Packard.)*

10.8.1 Antecedentes: la arquitectura CISC 80x86

La arquitectura Intel 80x86, que a fines de la década de 1990 se utiliza en cerca del 80% de las computadoras, tuvo sus raíces en el microprocesador 8086, desarrollado a fines de los años setenta. Las raíces arquitectónicas de la familia se remontan al procesador 8080 original de Intel, desarrollado a principios de los años setenta. Debido a su permanente apoyo a la teoría de la compatibilidad hacia delante, Intel ha sufrido de alguna manera

* *N. de T.*: A la fecha de la presente traducción, Intel anuncia el lanzamiento de IA-64 bajo el nombre comercial de Itanium. La información correspondiente puede encontrarse en el sitio http://developer.intel.com/design/itanium/index.htm.

con una arquitectura CISC que tiene más de 20 años de edad. Otros fabricantes, como Motorola, optaron por abandonar la compatibilidad del hardware en aras de la modernización, dejando a los emuladores la responsabilidad de simplificar la transición hacia una nueva arquitectura de programación.

En todo caso, Intel y Hewlett Packard plantearon ya hace algunos años que la arquitectura x86 llegaría prontamente al final de su vida útil, por lo que comenzaron el desarrollo conjunto de una nueva arquitectura. Intel y Hewlett Packard han llegado a decir, según comentarios, que las arquitecturas RISC de alguna forma se han "quedado sin combustible", por lo que sus desarrollos se orientan en otras direcciones. El resultado de esta investigación lleva a la "Arquitectura Intel 64", o IA-64. El primer miembro de la familia se conoce con el nombre de **Merced**, derivado del Río Merced, cerca de San José, California.

10.8.2 El procesador Merced: una arquitectura EPIC

Si bien Intel no ha publicado detalles significativos de la arquitectura de programación del Merced, hace referencia a su arquitectura como una **computadora de instrucciones explícitamente paralelas** (EPIC, *Explicitly Parallel Instruction Computing*). Intel insiste en destacar que no se trata de una máquina VLIW, y que ni siquiera es una máquina LIW (*long instruction word*, palabra de instrucciones larga), quizás a partir de la mala reputación lograda por las máquinas VLIW. No obstante, algunos analistas de la industria se refieren a ella como una "arquitectura EPIC del tipo de las VLIW".

Características

Si bien a la fecha de esta publicación no se conocen los detalles exactos, según las fuentes que han publicado información se espera que Merced ofrezca las siguientes características:

- Ciento veintiocho registros de uso general de 64 bits cada uno y, probablemente, 128 registros de punto flotante de 80 bits cada uno.
- Sesenta y cuatro registros de predicado de un bit cada uno (serán explicados más adelante).
- Palabras de instrucción que contienen tres instrucciones reunidas en un único paquete de 128 bits.
- Unidades de ejecución, del tipo de las unidades enteras, de punto flotante y de procesamiento de saltos, que aparecen en múltiplos de tres, en las que la arquitectura del IA-64 podrá repartir las instrucciones a ejecutar.
- Será tarea del compilador ordenar las instrucciones para aprovechar las ventajas de la existencia de múltiples unidades de ejecución.
- La mayor parte de las instrucciones tendrán aspecto similar a las de las arquitecturas RISC, si bien se comenta que el procesador podrá (¡todavía!) ejecutar códigos de 80x86, en una unidad de ejecución dedicada conocida como DXU.

- Carga especulativa. El procesador será capaz de cargar valores desde memoria bastante antes de su necesidad real. Las excepciones producidas por las cargas se posponen hasta que la ejecución llegue al lugar en que la carga debería haberse efectuado realmente.
- Anticipo (no predicción). Significa la ejecución de ambas salidas de una instrucción de salto condicional y el posterior descarte de los resultados correspondientes a la salida no tomada.

Estas últimas dos características se analizarán con más detalle.

La palabra de instrucción

La palabra de instrucción, de 128 bits, que se muestra en la figura 10.13, contiene tres instrucciones de 40 bits cada una y una plantilla de 8 bits. El compilador coloca esta plantilla para indicarle a la CPU qué instrucciones dentro de y cerca de esa palabra de instrucción pueden ejecutarse en paralelo, de donde surge el concepto de "explícito". La CPU no necesita analizar el código en tiempo de ejecución para descubrir cuáles son las instrucciones que pueden ejecutarse en paralelo debido a que el proceso de compilación lo ha determinado previamente. Los compiladores de la mayoría de las máquinas VLIW deben colocar instrucciones NOP en aquellos lugares en que no se admite la ejecución de instrucciones en paralelo. En el esquema de IA-64, la presencia de la plantilla identifica aquellas instrucciones de la palabra que pueden o que no pueden ejecutarse en paralelo, de modo tal que el compilador tiene libertad para acomodar instrucciones en la totalidad de las tres posiciones, independientemente de si se pueden ejecutar en paralelo.

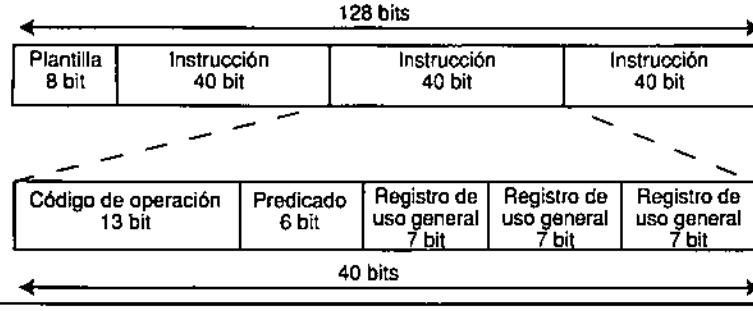


Figura 10.13 • La palabra de instrucción de IA-64 tiene 128 bytes.

El campo de predicado de 6 bits que forma parte de cada instrucción representa una identificación que ha sido colocada por el compilador para indicar si la instrucción es parte de alguna rama de una instrucción de salto condicional, y se utiliza para anticipar los saltos.

Anticipo de saltos

En lugar de realizar una predicción de los saltos, la arquitectura IA-64 se anticipa a los mismos con el objeto de evitar las penalidades derivadas de un salto mal previsto. Cuando el compilador detecta una instrucción de salto condicional a la que considera como candidata a ser anticipada, elige dos rótulos únicos e identifica las instrucciones que se encuentran en cada rama de la instrucción de salto con uno de esos dos rótulos, identificando también la rama a la que pertenecen. Ambas ramas del salto pueden ejecutarse en paralelo. Existen 64 registros de predicado de un bit, que corresponden cada uno de ellos a uno de los 64 posibles identificadores de anticipación.

Cuando se conoce la salida real del salto, se coloca en 1 el bit correspondiente del registro de predicado cuya salida del salto es verdadera, en tanto que se coloca un 0 en el registro de predicado de la salida de valor falso. Por consiguiente, se mantienen los resultados provenientes de las instrucciones que corresponden al rótulo de predicado correcto, en tanto que se descartan los resultados de las instrucciones que provienen del rótulo incorrecto.

Cargas especulativas

La arquitectura también utiliza **cargas especulativas** (esto es, examinar la secuencia de instrucciones para detectar futuras instrucciones de carga, y cargar el valor con anticipación, especulando con que dicho valor será realmente necesario y que no resultará alterado por las operaciones intervinientes). Si el resultado es exitoso, esto elimina la latencia normal inherente a los accesos a memoria. El compilador examina el flujo de instrucciones para seleccionar las operaciones de carga que puedan ser arrastradas hacia una posición anterior en la secuencia de instrucciones. En el lugar en que se encontraba originalmente la instrucción de carga deja una instrucción de verificación (*check*). Cuando se encuentra la instrucción de verificación, el valor del dato requerido ya se halla disponible en la CPU.

Uno de los problemas del esquema de cargas especulativas es que la operación de carga puede generar algún tipo de excepción (por ejemplo, porque la dirección no es válida). No obstante, es posible que esa excepción no sea genuina, debido a que la carga puede estar más allá de una instrucción de salto que no se ejecute, por lo que la carga en sí no se ejecutaría nunca. La arquitectura IA-64 pospone el procesamiento de las excepciones hasta el momento en que detecta la instrucción de verificación. Si no se ejecuta el salto, la instrucción de verificación tampoco se ejecuta y, en consecuencia, no se procesa la excepción.

Toda esta complejidad coloca sobre el compilador una carga pesada, lo que requerirá de un compilador ingenioso en cuanto a la forma en que se acomodan las operaciones en las palabras de instrucción.

Compatibilidad con 80x86

No hace mucho, Intel obtuvo una patente referida a un método para sostener dos juegos de instrucciones, uno de los cuales es el del procesador x86, método que se presume será utilizado en la arquitectura IA-64. Describe instrucciones para poder conmutar entre dos modos de ejecución y para compartir datos entre ellos.

Rendimiento estimado

Se ha estimado que la primera implementación de Merced aparecería en algún momento del año 2000, y tendría una frecuencia de reloj de 800 MHz. Según los objetivos que se plantean para el procesador, se espera que su rendimiento sea varias veces mejor que el de los procesadores de la generación actual cuando funcionan en modo EPIC y que el de un procesador Pentium II de 500 MHz funcionando en modo x86. Según Intel, las primeras aplicaciones del procesador IA-64 tendrán uso en estaciones de trabajo y servidores de alto rendimiento, y dada la estimación de costos unitarios del orden de $ 5.000 no cabe duda de que ese es el camino.

Por otra parte, los escépticos, que parecen abundar cuando se plantea la aparición de una nueva tecnología, dicen que la tecnología en cuestión no podrá satisfacer las expectativas y que probablemente el IA-64 podría no ver nunca la luz del día. El tiempo lo dirá.

10.9 Arquitecturas paralelo

Una forma de mejorar el rendimiento de un procesador consiste en disminuir el tiempo requerido para la ejecución de sus instrucciones. Esto funciona hasta un límite que está en el orden de los 400 MHz (Stone y Cocke), luego del cual los efectos conocidos como ***ringing*** impedirán un aumento de velocidad mayor, al menos con las tecnologías de bus convencionales. Esto no significa que no puedan admitirse frecuencias de reloj mayores, dado que de hecho los microprocesadores de la actualidad tienen frecuencias de reloj que están muy por encima de los 400 MHz. Lo que significa es que las soluciones de "bus compartido" se vuelven poco prácticas a esas frecuencias. A medida que disminuyen las posibilidades de mejorar el rendimiento mediante soluciones arquitectónicas convencionales, se hace necesario considerar métodos alternativos para lograrlo.

Una solución alternativa, que permite aumentar la velocidad de un bus, es el incremento de la cantidad de procesadores para descomponer y distribuir un programa único entre todos ellos. Esta solución, de **procesamiento paralelo**, hace que una cantidad de procesadores dada trabaje en forma conjunta, en paralelo, sobre un problema común. En el capítulo referido a la segmentación se ha planteado ya un ejemplo de procesamiento paralelo. En ese caso, se tienen cuatro procesadores conectados en serie (figura 10.3), cada uno de los cuales realiza una tarea diferente, como ocurre en una línea

de producción en una fábrica. La memoria entrelazada descripta en el capítulo 7 es también un ejemplo de segmentación.

Una arquitectura paralelo se puede caracterizar sobre la base de tres parámetros: (1) la cantidad de **elementos de procesamiento**, (2) la red de interconexión entre los citados elementos de procesamiento y (3) la organización de la memoria. En la segmentación de cuatro etapas de la figura 10.3, existen cuatro elementos de procesamiento. La red de interconexión entre los mismos es un simple **anillo**. La memoria es una memoria convencional de acceso aleatorio que se encuentra fuera del *pipeline*.

La caracterización de la arquitectura de un procesador paralelo es una tarea relativamente fácil, pero medir su rendimiento no es tan simple. Si bien se puede medir fácilmente el incremento de velocidad que ofrece una mejora simple como una estructura de segmentación, el aumento total de la velocidad depende de los datos; no todos los programas ni todos los conjuntos de datos encajan bien en una arquitectura segmentada. Otras consideraciones referidas al rendimiento de arquitecturas segmentadas, que también dependen de los datos, son el costo de limpiar un *pipeline*, el costo adicional de superficie, la latencia (retardo entre entrada y salida) de la segmentación, etcétera.

Entre las mediciones comunes de rendimiento se encuentran el **tiempo paralelo**, el **aumento de velocidad** (*speedup*), la **eficiencia** y la **respuesta** (*throughput*). El tiempo paralelo es sencillamente el tiempo total absoluto que requiere un programa para su ejecución en un procesador paralelo. El aumento de velocidad se determina como la relación entre el tiempo que se requiere para la ejecución del programa en un procesador secuencial (no paralelo) y el tiempo que se requiere para la ejecución del mismo programa en un procesador paralelo. En forma simple, se puede representar este aumento de velocidad (S, ahora en el contexto del procesamiento paralelo) como

$$S = \frac{T_{\text{Secuencial}}}{T_{\text{Paralelo}}}$$

Las programaciones de un mismo algoritmo en su forma secuencial y en su forma paralela pueden resultar muy distintas en diferentes máquinas, por lo que se requiere definir $T_{\text{Secuencial}}$ y T_{Paralelo} de modo que se pueda utilizar la mejor implementación en cada máquina.

La historia continúa. Si se requiere aumentar la velocidad de procesamiento en 100 veces, no es suficiente con distribuir el programa a través de 100 procesadores. El problema es que no hay demasiadas operaciones que puedan ser descompuestas en forma simple para utilizar completamente las unidades de procesamiento disponibles. Aun cuando haya algunas pocas operaciones secuenciales en un programa paralelo, el aumento de velocidad puede verse limitado en forma significativa. Esto se resume en la **ley de Amdahl**, que expresa el aumento de velocidad en función de la cantidad p de procesadores y la cantidad f de operaciones (tomada como fracción) que deben ejecutarse en forma secuencial.

$$S = \frac{1}{f + \dfrac{1-f}{p}}$$

Por ejemplo, si se requiere ejecutar en forma secuencial el 10% de las operaciones (f = 10%), el aumento de velocidad no puede ser mayor que 10 veces, no importa cuántos procesadores se utilicen:

$$S = \frac{1}{0{,}1 + \frac{0{,}9}{10}} \cong 5{,}3 \qquad S = \frac{1}{0{,}1 + \frac{0{,}9}{\infty}} = 10$$

p = 10 procesadores $\qquad$ $p = \infty$ procesadores

Este concepto lleva, a su vez, a la medida de la **eficiencia**. Se define como eficiencia a la relación entre el aumento de velocidad y la cantidad de procesadores utilizados. Para un aumento de velocidad de 5,3 con 10 procesadores, la eficiencia es de 5,3/10 = 0,53 = 53%.

Si se duplica la cantidad de procesadores a 20, el aumento de velocidad pasa a ser de 6,9, pero la eficiencia se reduce al 34%. En consecuencia, convertir un algoritmo en paralelo puede mejorar el rendimiento hasta un límite que queda determinado por la cantidad de operaciones secuenciales. La eficiencia se ve reducida en forma importante a medida que se llega a los límites admisibles para el aumento de velocidad, por lo tanto, no tiene sentido utilizar más procesadores en un cálculo con la idea de lograr una ganancia adicional en su rendimiento.

La **respuesta** (*throughput*) o producto de un sistema es una medición de la cantidad de cálculo realizada en el tiempo, y es de importancia considerable en aplicaciones segmentadas y vinculadas con operaciones de entrada-salida. Si se trata de un *pipeline* de cuatro etapas que se mantiene completo, caso en el cual cada etapa del sistema segmentado concluye su tarea en 10 ns, el tiempo medio para completar una operación es de 10 ns aun cuando lleva 40 ns ejecutar cualquier operación individual. La salida total en esta situación es entonces de

0,1 operaciones/ns = 10^8 operaciones por segundo

10.9.1 La taxonomía de Flynn

La arquitectura de una computadora puede clasificarse en función de sus **flujos de datos y de instrucciones**, usando una taxonomía desarrollada por M. J. Flynn. Un procesador secuencial convencional entra en la categoría de **procesadores con un único flujo de datos y un único flujo de instrucciones** (SISD, *single instruction stream, single data stream*), de acuerdo con lo que ilustra la figura 10.14.a. En un procesador SISD se ejecuta una única instrucción en cada instante de tiempo, si bien la segmentación permite que varias instrucciones se encuentren, en un mismo instante de tiempo, en distintas fases de su ejecución.

En un **procesador con un único flujo de instrucciones y múltiples flujos de datos** (SIMD, *single instruction stream, multiple data stream*), varios procesadores idénticos

realizan la misma secuencia de operaciones sobre distintos conjuntos de datos, según se muestra en la figura 10.14b. Un sistema SIMD puede imaginarse como una habitación llena de distribuidores de correo, donde todos clasifican distintas piezas de correo sobre el mismo conjunto de recipientes.

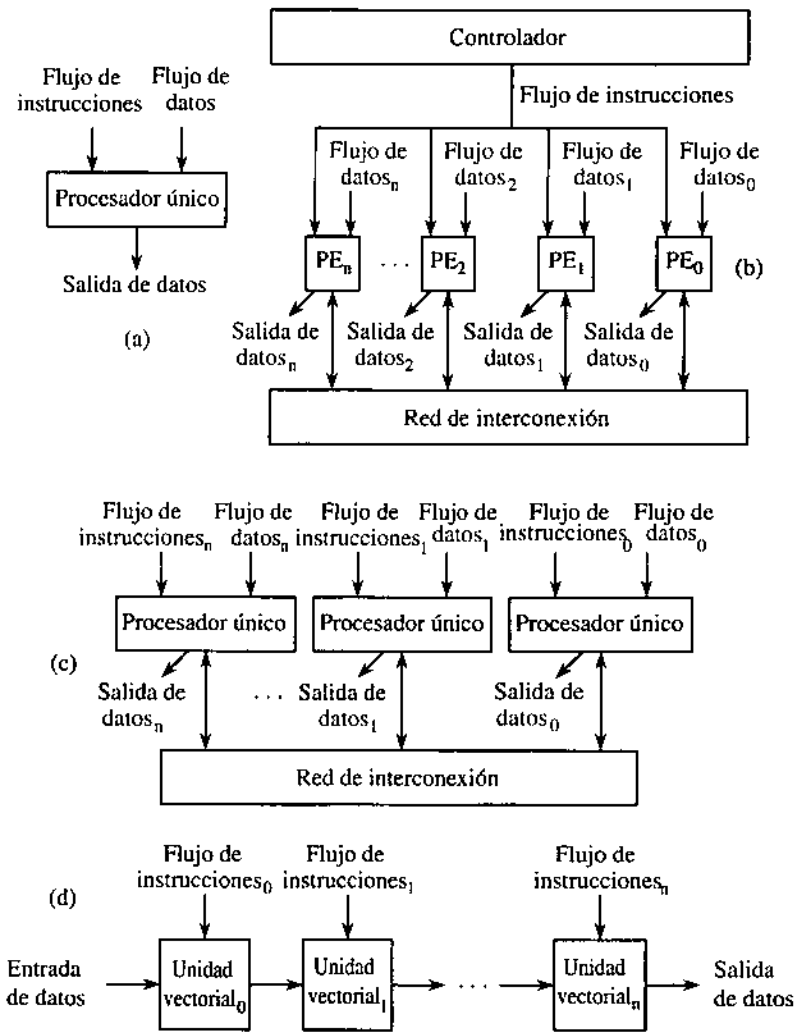


Figura 10.14 • Clasificación de arquitecturas de acuerdo con la taxonomía de Flynn: (a) SISD; (b) SIMD; (c) MIMD; (d) MISD.

En un **procesador con múltiples flujos de instrucciones y múltiples flujos de datos** (MIMD, *multiple instruction stream, multiple data stream*), varios procesadores distintos realizan operaciones distintas sobre conjuntos de datos también distintos, pero todos se hallan coordinados con el objeto de ejecutar un único programa paralelo, como puede verse en la figura 10.14c. Para ejemplificar el caso, se puede considerar el sistema de en-

tretenimiento familiar de video Sega, el que tiene cuatro procesadores para (1) síntesis de sonido (un procesador sintetizador Yamaha), (2) filtrado de sonido (un generador de sonido programable Texas), (3) ejecución de programas (un 68000 Motorola) y (4) procesamiento en segundo plano (un procesador Z80). Este sistema se analizará como caso de estudio al final del capítulo.

En un **procesador con múltiples flujos de instrucción y un único flujo de datos** (MISD, *multiple instruction stream, single data stream*), varias unidades funcionales actúan sobre un único conjunto de datos, según se puede observar en la figura 10.14d. El conjunto de datos suele ser, en una estructura típica, un vector de varios flujos de datos relacionados. Esta configuración se conoce como un **arreglo sistólico**, que ya fue analizado en el capítulo 3 bajo la forma de un multiplicador de arreglos.

10.9.2 Redes de interconexión

Cuando se distribuye una operación sobre varios elementos de procesamiento, estos necesitan comunicarse entre ellos a través de una **red de interconexión**. Existe una variedad de topologías de estas redes de interconexión, cada una de las cuales tiene sus propias características en términos **de complejidad de intersecciones** (una medida asintótica del área), **diámetro** (la longitud del trayecto a través de la red correspondiente al peor caso) y **bloqueo** (si se puede o no establecer una nueva conexión en presencia de otras conexiones). Esta sección describe algunas topologías representativas así como algunas estrategias de control para la configuración de redes.

Una de las topologías de red más poderosas es la topología *crossbar*, en la cual cada elemento de procesamiento se encuentra directamente conectado con cada uno de los demás elementos. La figura 10.15a muestra una imagen abstracta de esta topología, mediante la conexión de cuatro unidades de procesamiento. En una visión más cercana representada en la figura 10.16, se observa la existencia de **conmutadores de cruce** (*crosspoint switches*), dispositivos configurables que conectan o desconectan las líneas que lo atraviesan. En general, para N elementos de procesamiento, la complejidad medida en cantidad de intersecciones es N^2. En la figura 10.15a, $N = 4$ (no 8) debido a que los puertos de salida de los procesadores de la izquierda y los de entrada de los procesadores de la derecha pertenecen a los mismos elementos de procesamiento. La complejidad medida en cantidad de intersecciones es entonces $4^2 = 16$. El diámetro de la red es 1 debido a que cada elemento procesador puede comunicarse directamente con cada uno de los restantes, sin intermediarios. La cantidad de conmutadores de cruce que se atraviesan no se cuenta habitualmente para el cálculo del diámetro de la red. La estructura es **estrictamente no bloqueante**, lo que significa que siempre existe un trayecto disponible entre cada entrada y cada salida, independientemente de la configuración de los trayectos existentes.

En el otro extremo de la complejidad se encuentra la topología de bus, que se puede ver en la figura 10.15b. En esta topología, los distintos elementos procesadores comparten una cantidad fija de ancho de banda del bus. La complejidad de intersecciones para N

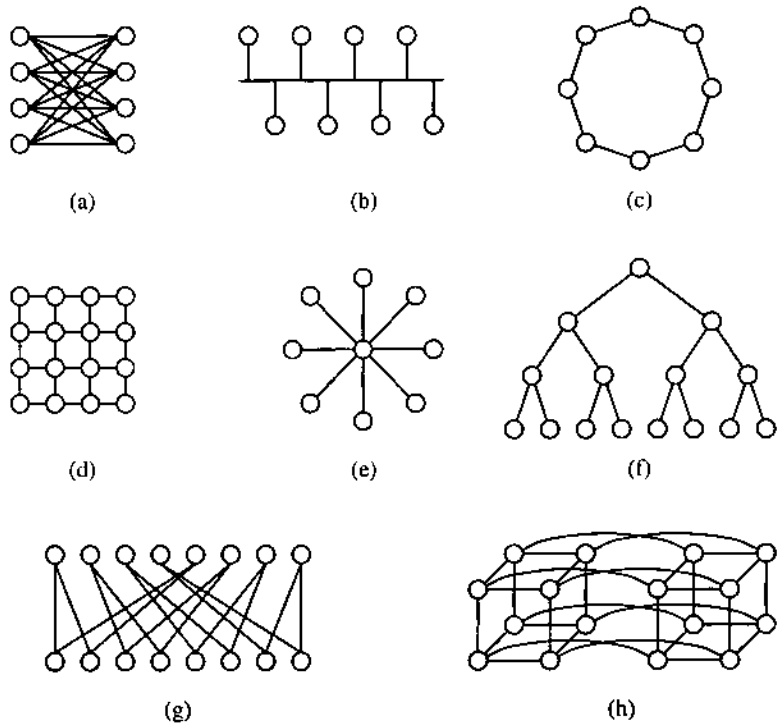


Figura 10.15 • Topologías de red: (a) crossbar; (b) bus; (c) anillo; (d) malla; (e) estrella; (f) árbol; (g) mezcla perfecta; (h) hipercubo.

procesadores es N y el diámetro de la red es 1, lo que significa que el crecimiento de la topología bus es más armónico que la topología *crossbar*. Puede haber un solo origen en cada instante de tiempo y, normalmente, hay un solo receptor, lo que significa que el bloqueo es una situación frecuente en este tipo de topología.

En una topología en anillo existen N intersecciones para N elementos de procesamiento, según se muestra en la figura 10.15c. Tal como en el caso del *crossbar*, cada intersección se encuentra incluida dentro de un elemento de procesamiento. El diámetro de la red es $N/2$, pero el ancho de banda colectivo es N veces mayor que en el caso de la topología en bus. Esto se debe a que los elementos procesadores adyacentes pueden comunicarse en forma directa sobre su vínculo común, sin afectar al resto de la red.

En la topología en malla, existen N intersecciones para N elementos procesadores, pero el diámetro es solamente $2\sqrt{N}$, de acuerdo con lo que se observa en la figura 10.15d. Todos los elementos procesadores pueden comunicarse simultáneamente en solo $3\sqrt{N}$ pasos, según surge del análisis realizado por Leighton, utilizando un **algoritmo de encaminamiento fuera de línea** (en el que la configuración de las intersecciones se determina fuera de los elementos procesadores).

En la topología en estrella existe un nodo central, como se ve en la figura 10.15e, a través del cual se comunican todos los elementos procesadores. Dado que toda la complejidad de la conexión se encuentra centralizada, la estrella solo puede crecer hasta tamaños determinados

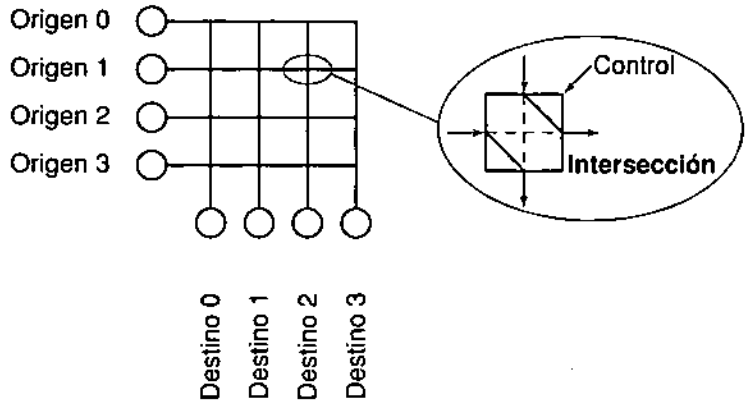


Figura 10.16 • Organización interna de un *crossbar*.

por la tecnología, lo que resulta normalmente menor que para el caso de topologías descentralizadas, como la topología en malla. La complejidad de las intersecciones dentro del nodo varía con la implementación, la que puede tomar cualquier aspecto, desde un *crossbar* hasta un bus.

En la topología en árbol existen N intersecciones para N procesadores, y el diámetro es $2log_2N - 1$, según se observa en la figura 10.15f. El árbol es efectivo en aplicaciones en las que hay gran cantidad de distribuciones y recolecciones de información.

En la topología de mezclado perfecto existen N intersecciones para N elementos procesadores, de acuerdo con la figura 10.15g. El diámetro es log_2N, dado que en el peor caso se requieren log_2N pasos para conectar un procesador con cualquier otro. El nombre viene de la propiedad que tiene un mazo de 2^N cartas, en el que N es un entero, el cual, si se lo corta al medio y se realizan N intercalaciones, recupera su configuración original. Todos los N elementos de procesamiento se pueden comunicar simultáneamente en $3log_2N - 1$ pasos a través de la red, de acuerdo con el análisis realizado por Wu y Feng.

Finalmente, la topología hipercubo tiene N intersecciones para N elementos procesadores, con un diámetro de $log_2N - 1$, según lo muestra la figura 10.15h. El menor número de cruces con respecto a la topología anterior se balancea con una mayor complejidad en la interconexión de los elementos procesadores.

A continuación, se considera el comportamiento de los bloqueos en redes de interconexión. La figura 10.17 ilustra una configuración en la que cuatro procesadores se interconectan a través de una red de mezclado perfecto de dos etapas, en la que cada intersección transfiere directamente ambas entradas a las salidas, o bien intercambia las entradas antes de enviarlas a las salidas. Se tiene un trayecto habilitado entre el procesador 3 y el procesador 0, y otro entre el procesador 0 y el procesador 3. Ni el procesador 1 ni el 2 requieren comunicarse, pero participan en algunas conexiones arbitrarias como efecto secundario de las configuraciones previamente especificadas para las intersecciones.

Supóngase ahora que se desea agregar alguna otra conexión, por ejemplo, entre el procesador 1 y el procesador 1. No hay forma de ajustar las intersecciones no utilizadas

para acomodar esta nueva conexión debido a que todas las intersecciones ya están establecidas, y la conexión requerida no ocurre como efecto secundario de las configuraciones existentes. Por consiguiente, la conexión 1 → 1 está bloqueada.

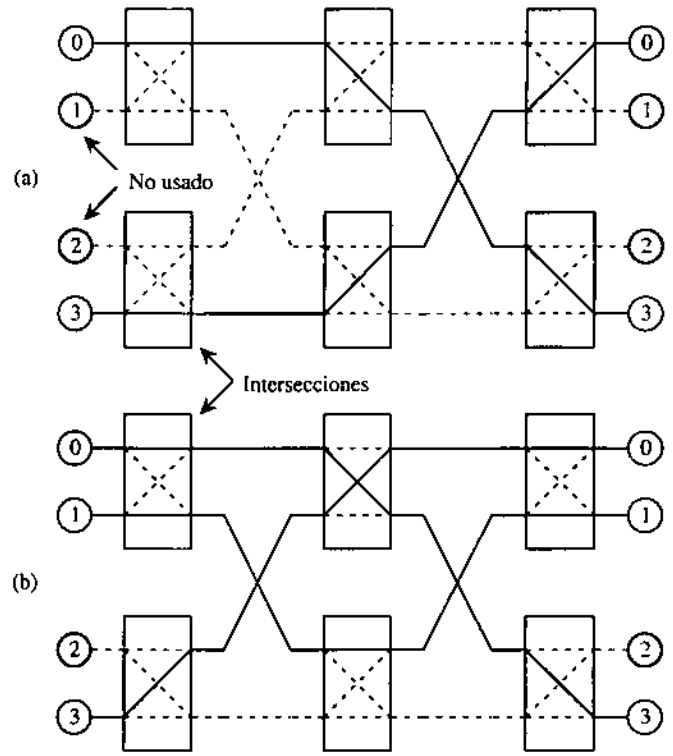


Figura 10.17 • (a) Configuración de intersecciones para las conexiones 0 → 3 y 3 → 0; (b) configuraciones ajustadas para acomodar la conexión 1 → 1.

Si se pudiesen modificar las configuraciones de las intersecciones en uso, se podrían acomodar las tres conexiones, tal como se muestra en la figura 10.17b. Una red de interconexión que funciona de esta manera se denomina **red no bloqueante por reconfiguración**.

La **red de Clos** de tres etapas es **estrictamente no bloqueante**. No hay necesidad de modificar las configuraciones de las intersecciones con el objeto de sumar alguna otra conexión. En la figura 10.18 se muestra un ejemplo de una red de Clos de tres etapas y cuatro elementos procesadores. En la etapa de entrada, cada intersección es en realidad un *crossbar* que permite realizar cualquier conexión de las dos entradas a las tres salidas. Las intersecciones en las etapas intermedia y de salida son también pequeñas estructuras *crossbar*. La cantidad de entradas a cada intersección de entrada y la cantidad de salidas desde cada intersección de salida se seleccionan de acuerdo con la complejidad deseada en las intersecciones y en la etapa intermedia.

La etapa intermedia tiene en este ejemplo tres intersecciones y, en general, se tienen $(n - 1) + (p - 1) + 1 = n + p - 1$ intersecciones en la etapa intermedia, en la que n es la

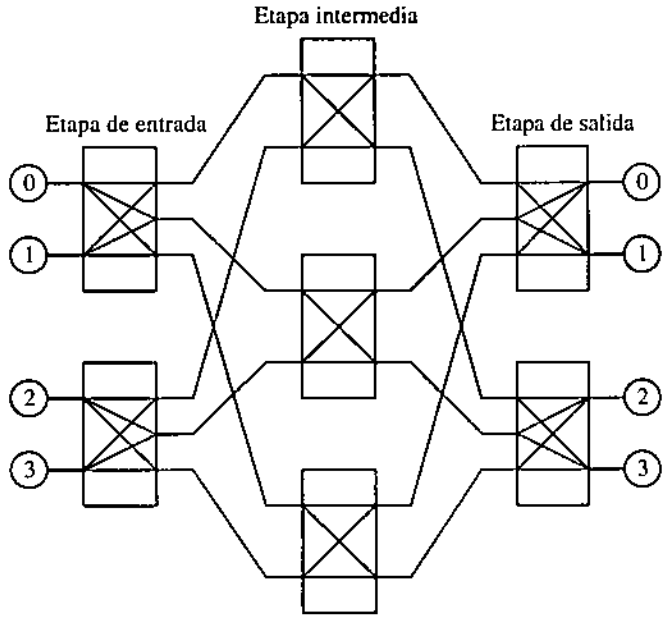


Figura 10.18 • Una red de Clos de tres etapas para cuatro elementos procesadores.

cantidad de entradas en cada intersección de entrada, en tanto que p es la cantidad de salidas desde cada intersección de salida. La forma de mantener una estructura estrictamente no bloqueante en una red de Clos de tres etapas es la siguiente: existen $n - 1$ formas de bloquear una entrada en la salida de una intersección de una etapa de entrada como resultado de conexiones existentes. En forma similar, existen $p - 1$ formas en las que las conexiones existentes pueden bloquear una conexión deseada hacia una intersección de salida. Para asegurar que pueda llevarse a cabo cualquier conexión deseada entre los puertos de entrada y salida disponibles, debe haber un trayecto adicional habilitado.

Para este caso, $n = 2$ y $p = 2$, por lo que se requieren $n + p - 1 = 3$ trayectos desde cada intersección de entrada hacia cada una de las intersecciones de salida. Arquitectónicamente, esta relación se satisface con tres intersecciones en la etapa intermedia, cada una de las cuales conecta cada intersección de entrada con cada intersección de salida.

Ejemplo: red estrictamente no bloqueante

En este ejemplo se desea diseñar una red estrictamente no bloqueante (Clos de tres etapas) para 12 canales (12 entradas y 12 salidas a la red), manteniendo baja la complejidad máxima de cualquier intersección de la red.

Existen distintas formas de organización de la red. Para la etapa de entrada, se pueden tener dos nodos de entrada con seis entradas por nodo, o seis nodos de entrada con dos

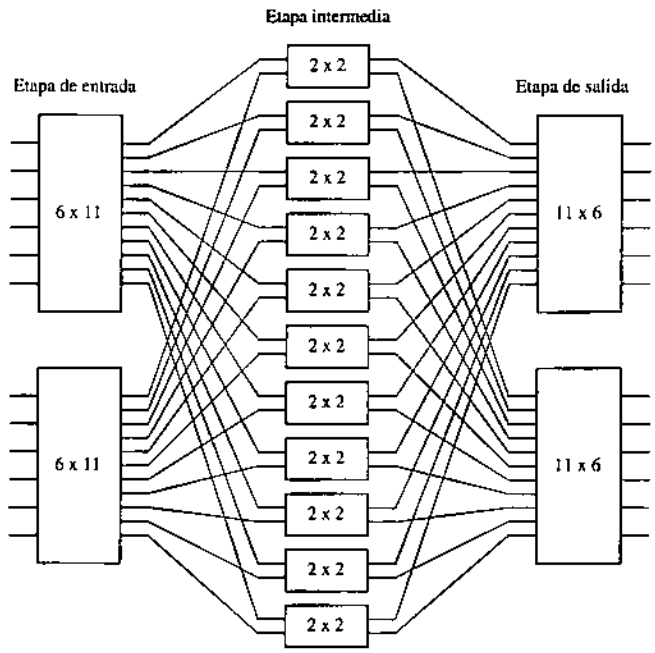


Figura 10.19 • Una red de Clos de tres etapas y 12 canales con n = p = 6.

entradas por nodo, por citar solo dos posibilidades. Existen alternativas similares en la etapa de salida. Se iniciará el análisis a partir de una configuración con dos nodos en la etapa de entrada y dos nodos en la etapa de salida, con seis entradas en cada nodo en la etapa de entrada y seis salidas en cada nodo de la etapa de salida. En este caso $n = p = 6$, lo que significa que se requieren $n + p - 1 = 11$ nodos en la etapa intermedia, como se muestra en la figura 10.19. La complejidad máxima de cualquier nodo es, en este caso, de 6 x 11 = 66, para cada uno de los nodos de entrada o de salida.

Si se realiza el intento con seis nodos de entrada y seis nodos de salida, con dos entradas en cada nodo de entrada y dos salidas en cada nodo de salida, se tiene $n = p = 2$, lo que implica que se requieren $n + p - 1 = 3$ nodos en la etapa intermedia, como se ve en la figura 10.20. La complejidad máxima de los nodos, en este caso, es de 6 x 6 = 36, lo que representa un valor mejor que la complejidad máxima de 66 del caso anterior.

En forma similar, las figuras 10.21 y 10.22 ilustran las redes configuradas con $n = p = 4$ y con $n = p = 3$, respectivamente. La mayor complejidad para cada una de estas redes es, respectivamente, de 4 x 7 = 28 y de 4 x 4 = 16. De las cuatro configuraciones aquí planteadas, $n = p = 3$ ofrece la menor de las máximas complejidades en los nodos. •

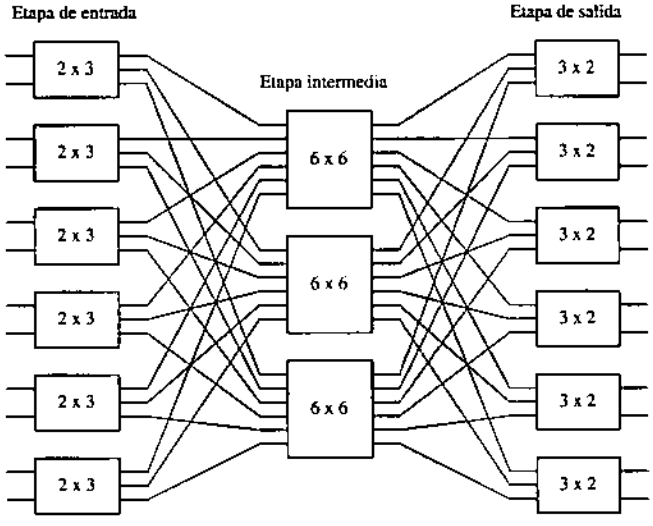


Figura 10.20 • Una red de Clos de tres etapas y 12 canales con $n = p = 2$.

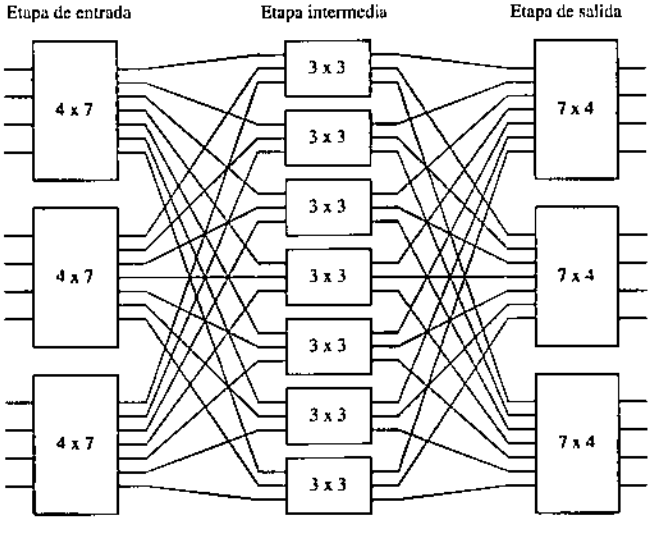


Figura 10.21 • Una red de Clos de tres etapas y 12 canales con $n = p = 4$.

10.9.3 Asignación de un algoritmo a una arquitectura paralelo

El proceso de asignación de un algoritmo a una arquitectura paralelo comienza con un **análisis de dependencias** en el que se identifican las dependencias de los datos entre las operaciones del programa. Se considerará el código del lenguaje C que se muestra en la figura 10.23. En un procesador SISD convencional, las cuatro sentencias numeradas requieren cuatro pasos para completarse, según lo muestra la secuencia de con-

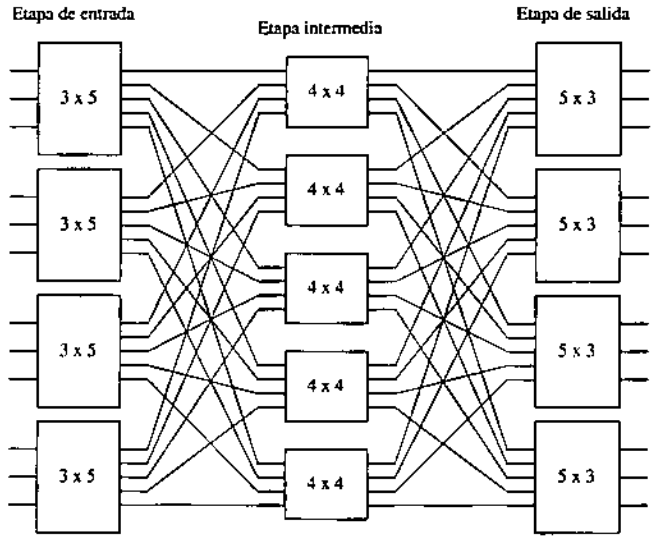


Figura 10.22 • Una red de Clos de tres etapas y 12 canales con $n = p = 3$.

trol de la figura 10.24a. El gráfico de dependencia de la figura 10.24b expresa el paralelismo natural de la secuencia de control. El **gráfico de dependencias** se crea asignando cada operación del programa principal a un nodo del gráfico, dibujando luego un arco dirigido desde cada nodo que produce un resultado hacia el (los) nodo(s) que lo necesiten.

Número de operación

```
func(x, y)  /* Calcular (x² + y²) × y² */
 int x, y;
 {
 int temp0, temp1, temp2, temp3;

0 temp0 = x * x;
1 temp1 = y * y;
2 temp2 = temp1 + temp2;
3 temp3 = temp1 * temp2;

 return(temp3);
 }
```

Figura 10.23 • Función del lenguaje C que calcula $(x^2 + y^2) \cdot y^2$.

La secuencia de control necesita cuatro pasos temporales para completarse, pero el gráfico de dependencia muestra que el programa puede completarse nada más que en tres pasos, dado que las operaciones 0 y 1 no dependen una de la otra, por lo que pueden ejecutarse en forma simultánea (en tanto haya dos procesadores disponibles). El aumento de velocidad resultante de $T_{Secuencial}/T_{Paralelo}$ 4/3 = 1,33... puede no ser demasiado grande, pero, como se podrá ver para otros programas, la posibilidad de aumentar la velocidad de procesamiento puede ser sustancial.

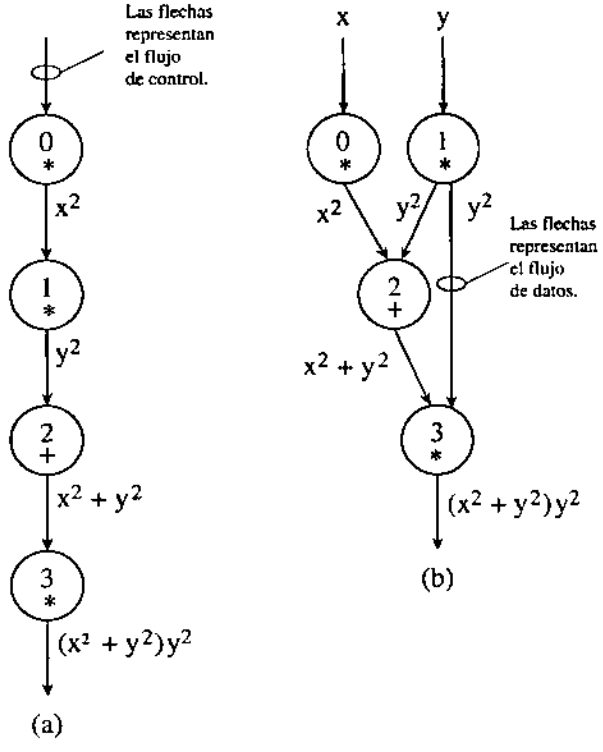


Figura 10.24 • (a) Secuencia de control del programa C; (b) gráfico de dependencias del programa C.

Considérese ahora el problema de multiplicar dos matrices $Ax = b$, en el que A es una matriz de 4 x 4 y tanto b como x son matrices de 4 x 1, como se ilustra en la figura 10.25a. El objetivo es resolver para b_i usando las ecuaciones de la figura 10.25b. A cada operación se le asigna un valor numérico, comenzando desde cero y terminando en 27. Hay 28 operaciones, si se asume que ninguna operación puede recibir más de dos operandos. Un programa para calcular b_i, que se ejecuta en un procesador SISD, requiere 28 pasos de tiempo para terminar, aceptando como simplificación que las sumas y los productos llevan el mismo tiempo en ejecutarse.

El gráfico de dependencia de este problema se muestra en la figura 10.26. En el peor caso, el trayecto desde cualquier entrada hasta cualquier salida atraviesa tres nodos, por lo tanto, todo el proceso se puede completar en tres pasos, lo que implica un aumento de velocidad de:

$$\frac{T_{\text{Secuencial}}}{T_{\text{Paralelo}}} = \frac{28}{3} = 9{,}33...$$

Ahora que se conoce la estructura de las dependencias, se puede plantear una asignación de los nodos del gráfico de dependencias a un conjunto de elementos de procesamiento en un procesador paralelo. La figura 10.27a ilustra una asignación en la que cada nodo del gráfico de dependencia de b_0 se asigna a un único elemento de procesamiento. El

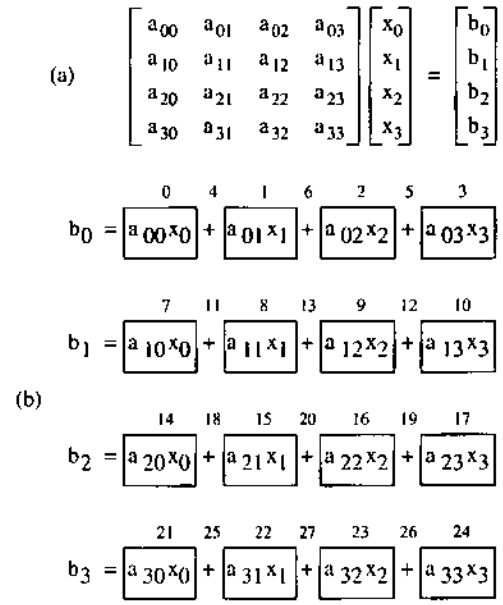


Figura 10.25 • (a) Configuración del problema para Ax = b; (b) las ecuaciones que calculan b_i.

tiempo requerido para completar cada suma es de 10 ns, en tanto que el tiempo requerido para completar cada producto es de 100 ns y el tiempo de comunicación entre elementos procesadores es de 1.000 ns. Estos números corresponden a un procesador ficticio, pero los métodos se pueden extender a procesadores paralelo reales.

Como se puede observar, del tiempo total requerido (2.120 ns) para ejecutar el programa usando la asignación de la figura 10.27a, el tiempo consumido en la comunicación domina el rendimiento. Este tiempo es peor que en el caso de una solución SISD, en la que los 16 productos y las 12 sumas requerirían 16 x 100 ns + 12 x 10 ns = 1.720 ns. En un procesador SISD no existe costo de comunicación entre procesadores, por lo tanto, solo debe considerarse el tiempo de cálculo.

Una asignación alternativa se muestra en la figura 10.27b, en la que todas las operaciones requeridas para el cálculo de b_0 se agrupan asignadas al mismo elemento procesador. Se ha aumentado la **granulación** del cálculo, que se define como una medida de la cantidad de operaciones asignadas a cada elemento procesador. Cada uno de los elementos procesadores es un procesador secuencial SISD, por consiguiente, ninguna de las operaciones dentro de un grupo pueden realizarse en paralelo. A cambio, el tiempo de comunicación entre procesadores se ha reducido a 0. Como se muestra en el diagrama, el tiempo paralelo para el cálculo de b_0 se ha reducido a 430 ns, lo que es mucho mejor que cualquiera de las dos soluciones anteriores, paralelo o SISD. Dado que no hay dependencias entre los b_i, todos ellos pueden calcularse en paralelo, usando un procesador por cada b_i. La mejora de velocidad es ahora de

$$\frac{T_{\text{Secuencial}}}{T_{\text{Paralelo}}} = \frac{1720}{430} = 4$$

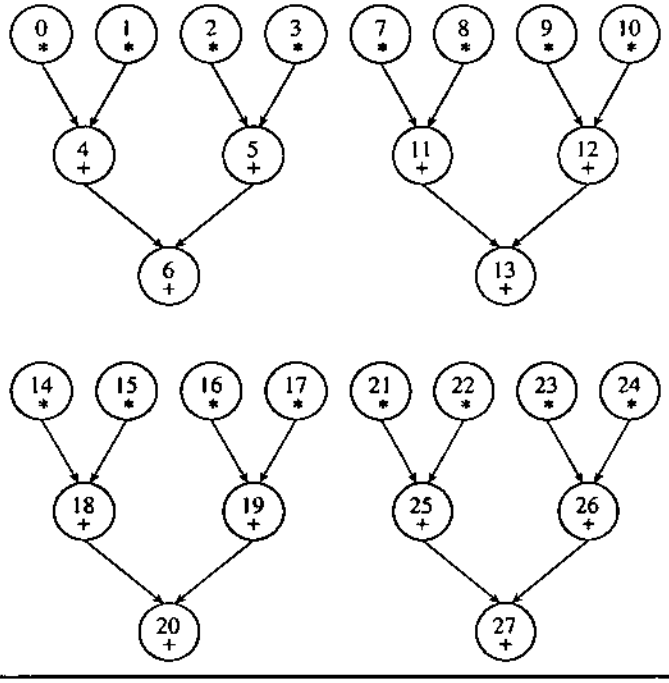


Figura 10.26 • Gráfico de dependencias para el producto de matrices.

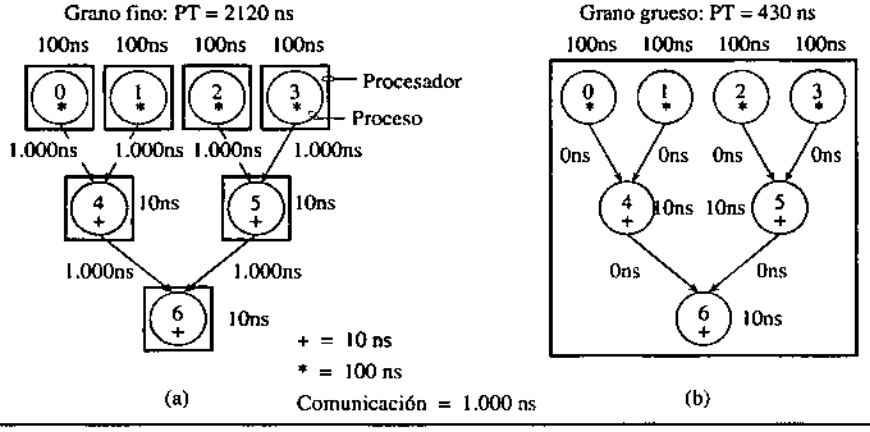


Figura 10.27 • Asignación de tareas a los elementos procesadores: (a) un procesador por operación; (b) un procesador por b_i.

La comunicación entre procesadores es siempre un cuello de botella en el procesamiento paralelo; por consiguiente, es muy importante mantener el balance apropiado. No se debe llegar a la confusión de pensar que el agregado de procesadores en un problema lo acelerará cuando, de hecho, el agregado de procesadores puede aumentar el tiempo de ejecución como resultado del tiempo de comunicación. En general, se pretende mantener una relación en la que

$$\frac{T_{\text{Comunicación}}}{T_{\text{Cálculo}}} \leq 1$$

10.9.4 Paralelismo de grano fino. La arquitectura CM-1

La arquitectura CM-1 corresponde a un procesador SIMD masivamente paralelo diseñado y construido por Thinking Machines Corporation durante la década de 1980. La arquitectura se plantea para resolver un alto nivel de conectividad entre una gran cantidad de procesadores de pequeño tamaño. CM-1 consiste en una gran cantidad de procesadores de 1 bit, distribuidos en los vértices de un hipercubo de n dimensiones. Cada procesador se comunica con cada uno de los demás a través de elementos de encaminamiento que envían y reciben mensajes a lo largo de cada dimensión del hipercubo.

La figura 10.28 ilustra un diagrama en bloques del procesador CM-1. La computadora principal es una máquina convencional SISD del tipo de la computadora Symbolics, popular en aquella época, la que corre un programa escrito en un lenguaje de alto nivel como LISP o C. Las partes del programa de alto nivel que pueden paralelizarse se envían hacia 2^n procesadores (el tamaño de una CM-1 completa es de 2^{16} procesadores) a través de un bus de memoria (para datos) y de un microcontrolador (para las instrucciones), en tanto que los resultados se recogen por medio del bus de memoria. Para la entrada y la salida se incluye un trayecto de datos separado, de gran ancho de banda, directamente hacia y desde el hipercubo.

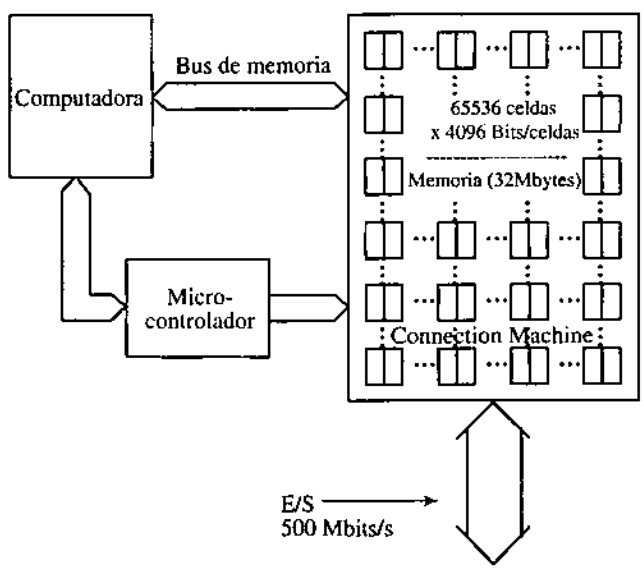


Figura 10.28 • Diagrama en bloques de CM-1 (adaptado de Hillis).

CM-1 hace uso de un hipercubo de 12 dimensiones entre los dispositivos de encaminamiento que reciben y envían paquetes de datos. El prototipo completo de CM-1 utiliza un hipercubo de 16 dimensiones, por lo que la diferencia entre el hipercubo de encaminamiento de 12 dimensiones y el espacio de 16 dimensiones de los elementos de procesamiento se plantea a través de un *crossbar* que sirva a los 16 elementos procesadores que se encuentran conectados a cada elemento encaminador. Para ejemplificar, en la figura

10.29 se muestra un hipercubo de cuatro dimensiones para la red de encaminamiento. Cada vértice del hipercubo es un elemento encaminador vinculado con un grupo de 16 elementos procesadores, cada uno de los cuales tiene una dirección binaria única. El hipercubo de la figura 10.29 sirve a 256 elementos procesadores. Se pueden determinar los encaminadores que se encuentran directamente conectados a otros encaminadores invirtiendo cualquiera de los cuatro bits más significativos de la dirección.

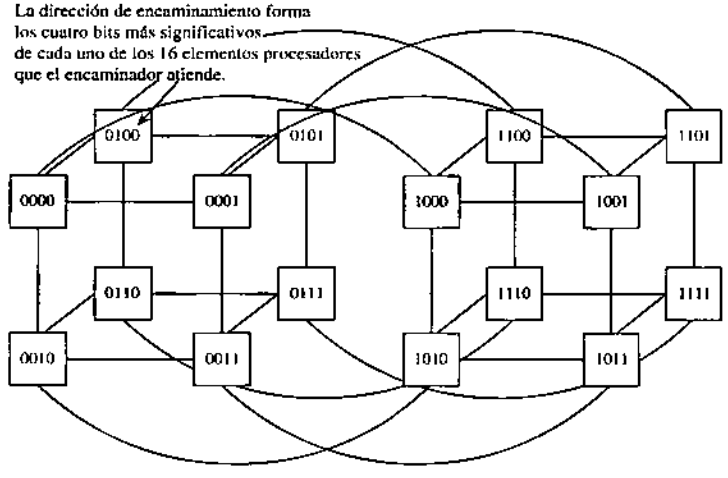


Figura 10.29 • Un hipercubo de cuatro dimensiones para la red de encaminamiento.

Cada elemento procesador se configura con un registro de banderas de 16 bits, una unidad aritmética de tres entradas y dos salidas, y una memoria de acceso aleatorio de 4.096 bits, como se ilustra en la figura 10.30. Durante la operación, un controlador externo (el microcontrolador de la figura 10.28) selecciona dos bits de memoria a través de las líneas de dirección A y B. Solo se puede leer una palabra desde memoria por vez, por lo que se retiene el valor A mientras se procede a buscar el valor B. El controlador selecciona una bandera a leer, y envía la bandera y los valores A y B a una unidad aritmético-lógica, cuya función también selecciona. El resultado del cálculo genera un nuevo valor para la posición correspondiente a la dirección A y una de las banderas.

La unidad aritmético-lógica toma tres entradas de datos de un bit, dos desde la memoria y una del registro de banderas, y 16 entradas de control desde el microcontrolador, produciendo dos salidas de datos de un bit para la memoria y los registros de banderas. La unidad aritmética genera todas las $2^3 = 8$ combinaciones (términos mínimos) de las variables de entrada para cada una de las dos salidas. Ocho de las 16 líneas de control seleccionan los términos mínimos requeridos en la forma canónica suma de productos de cada salida.

Los elementos procesadores se comunican con otros elementos procesadores a través de dispositivos encaminadores. Cada uno de ellos atiende la comunicación entre un elemento procesador y la red, para lo cual recibe los paquetes provenientes de la red desti-

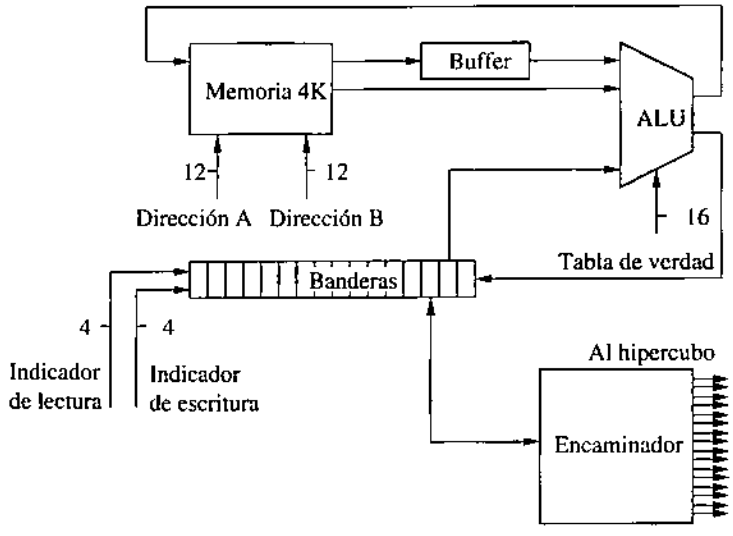


Figura 10.30 • Diagrama en bloques de un elemento procesador de CM-1.

nados a los procesadores adjuntos, inyecta paquetes en la red, nivelando velocidades cuando sea necesario, y reenvía los mensajes que usan al elemento encaminador como intermediario para llegar a su destino final.

CM-1 es una máquina de referencia cuya arquitectura convierte en realidad el paralelismo masivo. Cuando se trata de problemas escalables, como el **análisis de elementos finitos** (por ejemplo, la simulación del flujo calórico a través del uso de ecuaciones diferenciales parciales), el paralelismo disponible puede explotarse totalmente. En general, para este caso se requieren algunas operaciones de punto flotante, por lo que la generación siguiente, CM-2, ve aumentar la cantidad de elementos procesadores debido a los procesadores de punto flotante. Una forma natural de implementar un modelo del flujo calórico requiere una interconexión en malla, la que se implementa como una derivación cableada al mecanismo de encaminamiento de mensajes a través de la grilla Norte - Este - Oeste - Sur. De esta forma, se logra reducir el costo de las comunicaciones entre elementos procesadores.

No todos los problemas se pueden escalar en forma adecuada, y existe una tendencia general en el sentido de alejarse del procesamiento paralelo de grano fino. Esto se debe a la dificultad de mantener a los elementos procesadores ocupados haciendo algo útil y, simultáneamente, lograr que el tiempo de cálculo siga siendo mayor que el de comunicación. La sección siguiente analiza una arquitectura de grano grueso: CM-5.

10.9.5 Paralelismo de grano grueso: CM-5

El procesador CM-5 (Thinking Machines Corporation) combina propiedades de las arquitecturas SIMD y MIMD, por lo que ofrece una mayor flexibilidad para la asignación de un algoritmo paralelo a la arquitectura. La estructura de CM-5 se ilustra en la figura

10.31. Existen tres tipos de procesadores para el procesamiento de datos, el control y la entrada-salida. Estos procesadores se encuentran básicamente conectados por la red de datos y la red de control y, en menor medida, por la red de diagnóstico.

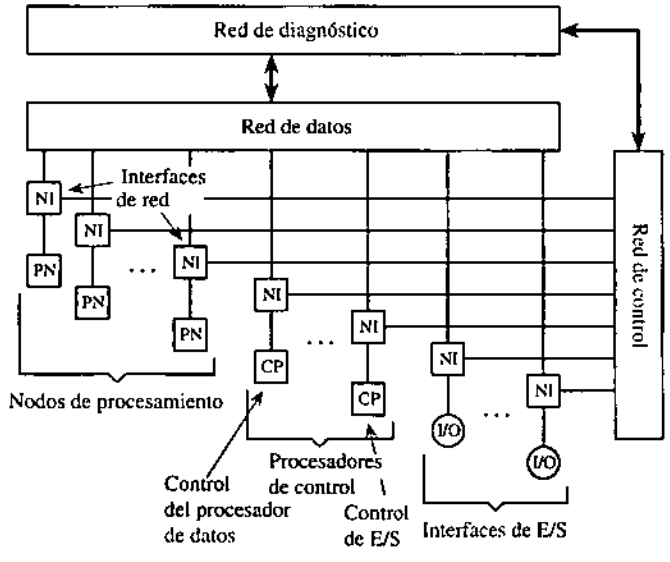


Figura 10.31 • La arquitectura de CM-5.

Los nodos de procesamiento se asignan a procesadores de control que forman **particiones**, según se muestra en la figura 10.32. Una partición incluye un procesador de control, una cantidad de nodos de procesamiento, y porciones dedicadas de las redes de control y de datos. Nótese la existencia tanto de particiones de usuario (en las que se lleva a cabo el procesamiento de los datos) como de particiones de entrada-salida.

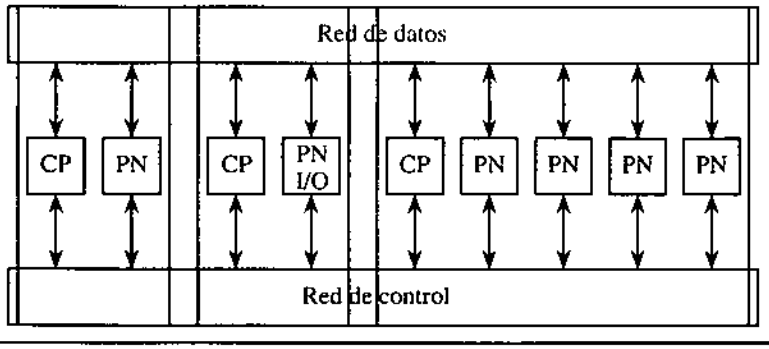


Figura 10.32 • Particiones en CM-5.

La red de datos utiliza una topología de **árbol grueso**, como la que se ilustra en la figura 10.33. La idea general es la del crecimiento del ancho de banda desde los nodos hijo hacia los nodos padre a medida que la red se acerca a su raíz, con el objeto de

contabilizar el tráfico que se incrementa a medida que los datos viajan desde las hojas hacia la raíz.

La red de control utiliza una topología de árbol binario simple, en la que los componentes del sistema están en las hojas. Un procesador de control ocupa una hoja de una partición, en tanto que los nodos procesadores se colocan en los nodos remanentes, aun cuando no se completen todas las posibles posiciones de los nodos en un subárbol.

La red de diagnóstico es un árbol binario separado que tiene uno o más procesadores de diagnóstico en su raíz. En las hojas hay componentes físicos, como placas de circuito, en lugar de componentes lógicos, como nodos de procesamiento.

Cada procesador de control es un sistema contenido en sí mismo, comparable en complejidad con una estación de trabajo. Un procesador de control contiene un microprocesador RISC que sirve como unidad central de proceso, una memoria local, elementos de entrada-salida que incluyen discos y conexiones Ethernet, y una interfaz CM-5.

Cada nodo de procesamiento es mucho menor y contiene un microprocesador basado en SPARC, un controlador de memoria para 8, 16 o 32 Mbytes de memoria local, y una interfaz de red hacia las redes de control y de datos. En una implementación completa de CM-5 puede haber hasta 16.384 nodos de procesamiento, cada uno de los cuales puede realizar operaciones enteras y de punto flotante de 64 bits, operando a una frecuencia de reloj de 32 MHz.

En su conjunto, CM-5 provee una combinación real de los estilos de procesamiento SIMD y MIMD, por lo que su grado de utilidad es mayor que el del estilo SIMD estricto de sus predecesores CM-1 y CM-2.

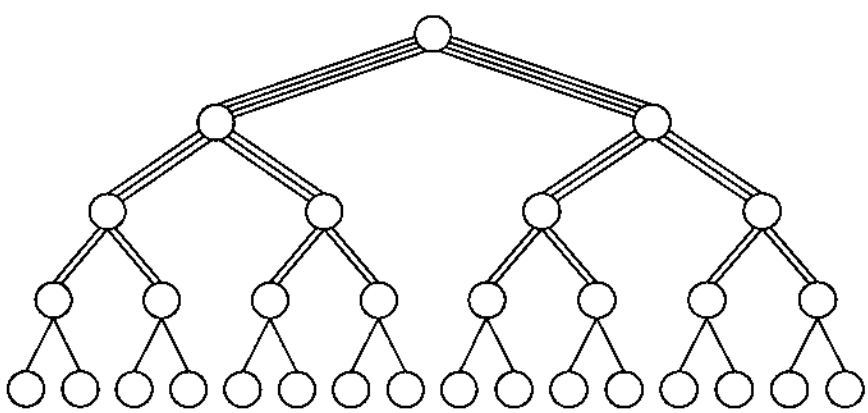


Figura 10.33 • Ejemplos de topología de árbol grueso.

10.10 Estudio de un caso: el procesamiento paralelo en Sega Génesis

Los sistemas hogareños de videojuegos son ejemplos de arquitecturas de computadoras casi completas. Tienen todas las características básicas de las arquitecturas de computadoras modernas y varias prestaciones avanzadas. Una falencia notable es la falta de un

dispositivo de almacenamiento permanente (como un disco rígido) para guardar información, aunque los modelos más actualizados también incluyen hasta cierto grado esta prestación. Entre sus características avanzadas debe destacarse el uso, aquí analizado, de múltiples procesadores en configuración MIMD.

Tres de las plataformas para videojuegos hogareños más importantes son las fabricadas por **Sony**, **Nintendo** y **Sega**. En este análisis se procederá a estudiar el caso de Sega Génesis, que utiliza procesamiento paralelo para su funcionamiento en tiempo real.

10.10.1 La arquitectura Sega Génesis

La figura 10.34 ilustra el aspecto externo del sistema de videojuego hogareño Sega Génesis. El mismo está compuesto por una placa madre (que contiene componentes electrónicos tales como el procesador, la memoria y los elementos de interconexión), por algunos controladores manuales y por una interfaz a un aparato televisor.

Sega Génesis contiene todos los elementos básicos requeridos por el modelo convencional de computadoras de von Neumann: entrada (los controladores), salida (el aparato televisor), unidad aritmético-lógica (en el procesador), unidad de control (también en el procesador) y memoria (la que incluye tanto la memoria interna como los cartuchos enchufables que contienen los juegos).

Figura 10.34 • Aspecto externo del sistema de videojuego hogareño Sega Génesis.

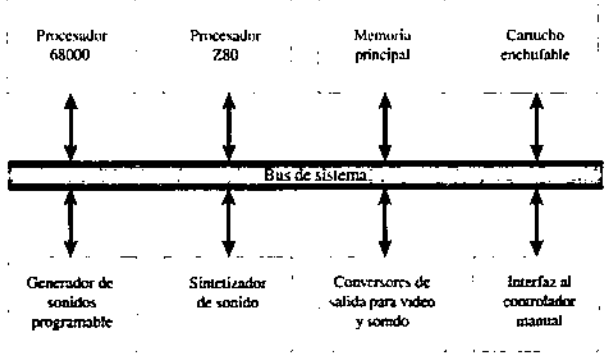


Figura 10.35 • Visión de Sega Génesis desde el modelo de bus del sistema.

El modelo de bus del sistema implementa la conectividad lógica de la arquitectura Sega así como parte de su organización física. La figura 10.35 ilustra el diagrama en bloques del sistema Sega Génesis. Génesis contiene dos microprocesadores de propósitos generales, un Motorola 68000 y un Zilog Z80. Estos procesadores, viejos y de bajo costo, manejan la ejecución general de los programas. Los sistemas de videojuegos deben ser capaces de generar una gran cantidad de efectos sonoros, proceso que hace uso intensivo de los recursos de cómputo. Con el objeto de mantener la calidad y la velocidad del juego mientras se generan los sonidos, el Génesis descarga los cálculos de los efectos de sonido hacia dos circuitos dedicados, un generador programable de sonidos de Texas Instruments (TI PSG) y un circuito sintetizador de sonidos Yamaha. Existen también interfaces de entrada y salida para el sistema de video y los controles manuales.

El procesador 68000 ejecuta el programa principal y controla al resto de la máquina, cumpliendo con esta función por medio de la transferencia de datos e instrucciones a los restantes componentes a través del bus del sistema. Uno de los componentes que el procesador 68000 controla es un procesador de arquitectura similar pero más pequeño, conocido como Z80, el que puede cargarse con un programa que se ejecuta mientras el 68000 vuelve a ejecutar su propio programa, utilizando un mecanismo de arbitraje que permite que ambos procesadores compartan el bus (pero solo uno por vez).

El generador de sonidos programable tiene tres tonos de onda cuadrada y un tono de ruido blanco. Cada tono/ruido puede tener su propia frecuencia y volumen.

El sintetizador Yamaha se basa en mecanismos de síntesis de modulación de frecuencia. Existen seis voces con cuatro operadores cada uno. El circuito es similar al que se utiliza en los sintetizadores Yamaha DX27 y DX100. Se logra crear una gran variedad de sonidos por medio de la programación de registros internos a los circuitos integrados.

Los cartuchos externos contienen los programas de juego, existiendo memoria adicional disponible para el tiempo de ejecución en una unidad separada (rotulada "Memoria principal", en la figura 10.35). Se proveen componentes adicionales para la salida de video, la salida de sonido y los controladores manuales.

10.10.2 Operación de Sega Génesis

En el momento del encendido inicial de Sega Génesis, se habilita una señal de RESET, la que permite la estabilización de todos los niveles de tensión eléctrica y la inicialización de una cantidad de variables requeridas en tiempo de ejecución. La señal de RESET se inhibe automáticamente, tras lo cual el 68000 comienza la lectura y ejecución de las instrucciones provenientes del cartucho de juego.

Durante la operación, las instrucciones del cartucho de juego hacen que el 68000 cargue un programa en el procesador Z80 y que comience la ejecución de dicho programa, mientras el 68000 regresa a seguir ejecutando su propio programa. El programa del Z80 controla los circuitos integrados de sonido, en tanto que el 68000 realiza las operaciones gráficas, verifica la actividad de los controladores manuales y ejecuta el programa general del juego.

10.10.3 Programación de Sega Génesis

(*Nota de los autores*: Esta sección se ha adaptado de un aporte realizado por David Ashley, dash@xdr.com.)

Sega Génesis utiliza cartuchos externos enchufables para almacenar los programas de juego. Por medio de proveedores externos es posible adquirir cartuchos en blanco, los que se pueden programar utilizando una **placa programadora de memorias PROM** que puede conectarse en una computadora de escritorio. Los juegos pueden programarse en lenguajes de alto nivel, que luego se compilan hacia lenguaje simbólico, o, directamente, se programan en lenguaje ensamblador (aún hoy, el lenguaje de máquina se sigue utilizando fuertemente en la programación de juegos). Un conjunto de herramientas de desarrollo traduce el lenguaje fuente en código objeto que puede grabarse directamente en los cartuchos (solo una vez en cada cartucho). Como alternativa a la programación de cartuchos, durante el desarrollo el cartucho puede reemplazarse por una tarjeta de desarrollo reprogramable.

Génesis contiene dos microprocesadores de propósitos generales, el Motorola 68000 y el Zilog Z80. El 68000 funciona a 8 MHz y contiene 64 Kb de memoria dedicada. El cartucho de memoria ROM aparece en la dirección de memoria 0. El 68000 descarga los cálculos referidos a los efectos de sonido sobre el generador programable de Texas Instruments y el sintetizador Yamaha.

El hardware gráfico de Génesis consiste en dos planos que se desplazan. Cada plano está compuesto por elementos, cada uno de los cuales es un cuadrado de 8 píxeles por lado, con 4 bits por píxel. Cada píxel puede tener así 16 colores. Cada elemento usa una de cuatro tablas de colores, por lo que en la pantalla pueden aparecer hasta 64 colores por vez, aun cuando solo puede haber 16 colores diferentes en cada uno de los elementos. Existen 64 KB de memoria para gráficos, los que permiten 2.048 elementos individuales si la memoria no se utiliza para otra cosa.

Cada plano puede desplazarse independientemente en varias formas. Los planos consisten en tablas de palabras, cada una de las cuales describe un elemento. Una palabra contiene 11 bits para identificar el elemento, 2 bits para "*flip x*" y "*flip y*", dos bits para la selección de la tabla de colores y un bit para un selector de profundidad. Los personajes (*sprites*) están compuestos por el mismo tipo de elementos. Cada uno puede estar formado por hasta cuatro elementos de alto y hasta cuatro de ancho. Dado que el tamaño de cada elemento es de 8 x 8, los personajes pueden tener cualquier tamaño desde 8 x 8 píxeles hasta 32 x 32 píxeles. Pueden existir hasta 80 personajes en pantalla en un momento dado. En una línea única de barrido puede haber 10 personajes de 32 píxeles de ancho o 20 de 16 píxeles de ancho. Cada personaje puede tener solo 16 colores, que deben tomarse desde las 4 tablas de color diferentes. Se asignan 3 bits de color para cada cañón, por lo que se logran 512 colores, siendo el color 0 la transparencia.

Existe un programa copiador de memoria, residente en hardware, que realiza copias rápidas desde la memoria del 68000 hacia la memoria gráfica. El Z80 también tiene 8 Kb

de memoria RAM. El Z80 puede acceder a los circuitos gráficos o a los de sonido, pero normalmente estos circuitos se encuentran controlados por el 68000.

El proceso de creación de un cartucho de juegos implica (1) escribir el programa del juego, (2) traducirlo a código objeto (compilación, ensamblado y enlace del código hacia un módulo objeto ejecutable; algunas partes del programa pueden escribirse en lenguaje de alto nivel y otras, directamente, en lenguaje de máquina), (3) verificar el funcionamiento del programa sobre una tarjeta de desarrollo reprogramable (si estuviese disponible) y (4) grabar el programa en un cartucho vacío.

En la sección "Para lectura posterior" existe más información acerca de la programación de Sega Génesis.

Resumen

En el enfoque RISC, las instrucciones que se ejecutan más frecuentemente se optimizan por medio de la eliminación o reducción de la complejidad de otras instrucciones y modos de direccionamiento que normalmente se encuentran en las arquitecturas CISC. El rendimiento de las arquitecturas RISC se aumenta aún más por medio de la segmentación y del incremento de la cantidad de registros disponibles para la CPU. Las estructuras superescalar y VLIW son ejemplos de mejoras de eficiencias que amplían, más que reemplazan, el criterio RISC.

Las arquitecturas paralelas se pueden dividir en MISD, SIMD o MIMD. La solución MISD se utiliza para el procesamiento sistólico de arreglos y es la arquitectura menos general de las tres. En una arquitectura SIMD, todas las unidades de proceso ejecutan las mismas operaciones en diferentes conjuntos de datos, en un enfoque tipo "ejército de hormigas" del procesamiento paralelo. La estructura MIMD puede caracterizarse como una "horda de elefantes", debido a que existe una pequeña cantidad de procesadores potentes, cada uno con su propio flujo de datos y de instrucciones.

La orientación actual es la del alejamiento del paralelismo de grano fino que se ejemplifica en las estructuras MISD y SIMD, orientándose hacia la alternativa MIMD. Esto se debe a los altos costos en tiempos de comunicación entre las unidades procesadoras y la economía que significa el uso de redes de estaciones de trabajo sobre procesadores paralelos rígidamente acoplados. El objetivo de la solución MIMD es mejorar el balance entre los tiempos consumidos en el cálculo y en las comunicaciones.

Para lectura posterior

Las tres características básicas de las arquitecturas RISC enumeradas en la sección 10.2 surgieron en el Centro de Investigaciones T. J. Watson de IBM, según fueron resumidos por A. Ralston y E. D. Reilly. La obra de J. L. Hennessy y D. A. Patterson es una referencia clásica con respecto a gran parte del trabajo que lleva al concepto RISC, aun cuando la palabra "RISC" no aparece en el título de su

libro. W. Stallings (1991) es una referencia completa sobre sistemas RISC. Y. Tamir y C. Sequin muestran que el tamaño de ventana de ocho se moverá en menos del 1% de los llamados o retornos. A. Tanenbaum hace una introducción legible al concepto RISC. C. Dulong describe el sistema IA-64. La arquitectura Power PC 601 se describe en la literatura de Motorola.

Las obras de M. J. Quinn y de K. Hwang analizan el campo del procesamiento paralelo en términos de arquitecturas y algoritmos. M. J. Flynn cubre la taxonomía de arquitecturas. T. Yang y A. Gerasoulis justifican el hecho de mantener la relación entre tiempos de comunicaciones y de cálculo en valores menores que 1. Tanto la obra de W. D. Hillis como el material publicado por Hillis y L. W. Tucker describen las arquitecturas de CM-1 y CM-5, respectivamente. J. Y. Hui analiza las redes de interconexión y F. T. Leighton, los algoritmos de encaminamiento para varios tipos de redes de interconexión. C. L. Wu y T. J. Feng cubren el encaminamiento en una red de intercambio desordenada.

La información adicional para programar el sistema Sega Génesis se puede encontrar en la dirección http://hiwaay.net/~jfrohwei/sega/genesis.html.

Dulong, C., "The IA-64 architecture at work", en: *IEEE Computer*, n° 31, julio de 1998, p.p. 24-32.

Flynn, M. J. "Some Computer Organizations and their Effectiveness", en: *IEEE Transactions on Computers*, vol. 30, n° 7, 1972, p.p. 948-960.

Hennessy, J. L y D. A. Patterson, *Computer Architectura: A Quantitative Approach*, 2ª ed., Morgan Kaufmann Publishers, 1995.

Hillis, W. D. y L. W. Tucker, "The CM-5 Connection Machine: A Scalable Supercomputer", en: *Communications of the ACM*, vol. 36, n° 11, noviembre de 1993, p.p. 31-40.

Hillis, W. D., *The Connection Machine*, MIT Press, 1985.

Hui, J. Y. *Switching and Traffic Theory for Integrated Broadband Networks*, Kluwer Academic Publishers, 1990.

Hwang, K., *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw Hill, 1993.

Knuth, D. E. "An empirical study of Fortran programs", en: *Software – Practice and Experience*, n° 1, 1971, p.p. 105-133.

Leighton, F. T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, 1992.

Motorola, Inc., *Power PC 601 RISC Microprocessor User's Manual*, Motorola Literature Distribution, P.O. Box 20912, Phoenix, AZ, 85036.

Quinn, M. J., *Designing Efficient Algorithms for Parallel Computers*, McGraw Hill, 1987.

Ralston, A. y E. D. Reilly, eds., *Encyclopedia of Computer Science*, 3ª ed., van Nostrand Reinhold, 1993.

SPARC International, Inc., *The SPARC Architecture Manual: Versión 8*, Prentice Hall, 1992.

Stallings, W., *Computer Organization and Architecture: Designing for Perfomance*, 4ª ed., Prentice Hall, 1996. (Traducción al español disponible: *Organización y arquitectura de computadores*, 5ª ed., Prentice Hall, 2000.)

Stallings, W., *Reduced Instruction Set Computers*, 3ª ed., IEEE Computer Society Press, 1991.

Stone, H. S y J. Cocke, "Computer architecture in the 1990s", en: *IEEE Computer*, vol. 24, n° 9, septiembre de 1991, p.p. 30-38.

Tamir, Y. y C. Sequin, "Strategies for managing the register file in RISC", en: *IEEE Transactions on Computers*, noviembre de 1983.

Tanenbaum, A., *Structured Computer Organization*, 4ª ed., Prentice Hall, 1999. (Traducción disponible al español: Organización *de computadoras*, 5ª ed., Prentice Hall, 2000.)

Wu, C. L. y T. Y. Feng, "The Universality of the Shuffle Exchange Network", en: *IEEE Transactions on Computers*, C-30, n° 5, 1981, p.p. 324-332.

Yang, T. y A. Gerasoulis, "A fast static Scheduling Algorithm for DAGs on an Unbounded number of processors", en: *Proceedings of Supercomputing '91*, Albuquerque, Nuevo México, noviembre de 1991.

Problemas

10.1 El incremento de la cantidad de ciclos por instrucción a veces puede mejorar la eficiencia de ejecución de un *pipeline*. Si el tiempo por ciclo del *pipeline* descripto en la sección 10.3 es de 20 ns, el CPI_{medio} es de 1,5 x 20 ns = 30 ns. Calcular la eficiencia de ejecución de la misma estructura cuando la profundidad del *pipeline* se incrementa de 5 a 6 y el tiempo de ciclo decrece de 20 ns a 10 ns.

10.2 El código SPARC que se describe a continuación se ha tomado del código generado por `gcc` en la figura 10.10. ¿Puede usarse `%r0` en cada una de las tres líneas en lugar de "gastar" `%r1` en la segunda línea?

```
    ...
st    %o0, [%fp-28]
sethi %hi(.LLC0), %o1
or    %o1, %lo(.LLC0), %o1
    ...
```

10.3 Calcular el aumento de velocidad que puede esperarse si se reemplaza un circuito Pentium de 200 MHz por otro de 300 MHz, si todos los demás parámetros se mantienen constantes.

10.4 ¿Cuál es el aumento de velocidad que puede esperarse si el conjunto de instrucciones de determinada máquina se modifica de modo tal que la instrucción de salto se ejecute en solo un ciclo de reloj en lugar de tres, considerando que las instrucciones de salto son el 20% del total de las instrucciones ejecutadas por un cierto programa? Suponer que las restantes instrucciones se ejecutan en un tiempo medio de tres ciclos de reloj por instrucción y que el cambio no altera ninguna otra cosa.

10.5 Crear un gráfico de dependencia para la expresión siguiente:

$f(x, y) = x^2 + 2xy + y^2$

10.6 Dados 100 procesadores para un cálculo que contiene un 5% de código que no puede ser paralelizado, calcular el aumento de velocidad y la eficiencia.

10.7 ¿Cuál es el diámetro de un hipercubo de 16 dimensiones?

10.8 Para el ejemplo que se encuentra al final de la sección 10.9.2, calcular la complejidad total de las intersecciones a lo largo de las tres etapas.

Apéndice A

Lógica digital

A.1 Introducción

El objeto de este apéndice es repasar algunos principios básicos de la lógica digital, que pueden aplicarse en el diseño de una computadora digital. Comienza con el estudio de la **lógica combinatoria**, aquella en la que las decisiones lógicas se toman exclusivamente sobre la base de las combinaciones de las entradas. Continúa con la **lógica secuencial**, en la que las decisiones se adoptan no solo en virtud de las combinaciones de las entradas actuales sino también de la historia anterior de las mismas entradas. Con el entendimiento de estos principios básicos, se pueden diseñar circuitos lógicos digitales a partir de los cuales construir íntegramente una computadora. El primer punto a estudiar es el bloque constructivo fundamental de una computadora digital, la **unidad lógica combinatoria**.

A.2 Lógica combinatoria

Una unidad lógica combinatoria transforma un conjunto de entradas en un conjunto de salidas de acuerdo con una o más funciones lógicas. Las salidas de un circuito combinatorio son rigurosamente función de las entradas, y se actualizan inmediatamente luego de cualquier cambio en las entradas. La figura A.1 ilustra un modelo de unidad lógica combinatoria. La misma recibe un conjunto de entradas i_0-i_n y produce un conjunto de salidas f_0-f_m, las que dependerán de las funciones lógicas correspondientes. En este tipo de circuito lógico combinatorio no existe realimentación de las salidas sobre las entradas (la sección A.11 analiza los circuitos con realimentación).

Las entradas y salidas de un circuito combinatorio adoptan dos valores diferentes: alto y bajo. En los circuitos que se conocen como **digitales**, las señales (valores) son parte de un conjunto finito de valores. Un circuito electrónico digital recibe entradas y genera salidas en las cuales es habitual considerar como valor bajo el de 0 Volt, en tanto que se adopta como valor alto el de 5 Volt. Esta convención no es de uso universal. En los circuitos de alta velocidad se tiende a usar menores valores de tensión. Algunos cir-

Figura A.1 • Diagrama en bloques de una unidad lógica combinatoria.

cuitos de computadora funcionan en el dominio **analógico**, en el que se admite una variación continua de las señales, y en el caso de los circuitos digitales ópticos se pueden utilizar variaciones de fase o polarizaciones, con lo que no tiene sentido plantear los conceptos de alto o bajo. Existen aplicaciones en las que resulta más apropiado el uso de circuitos analógicos, como los simuladores de vuelo, donde los circuitos analógicos se aproximan mucho más a los mecanismos de una aeronave que los circuitos digitales.

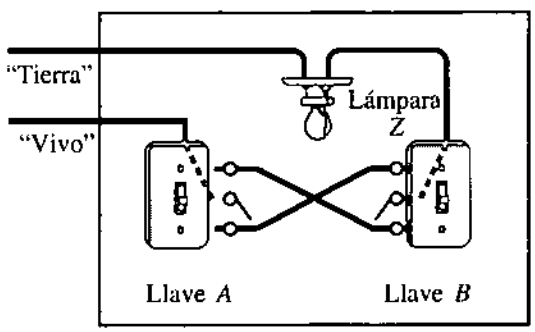
Si bien las computadoras digitales son binarias en su inmensa mayoría, también existen **circuitos de niveles múltiples**. Un cable capaz de transportar más de dos valores puede ser mucho más eficiente en la transmisión de datos que un cable que solo transporta dos niveles. Un circuito digital de varios niveles no es similar a un circuito analógico, dado que el circuito digital, en este caso, trata con señales que pueden adoptar uno de un grupo finito de valores, en tanto que una señal analógica maneja un continuo de valores. El uso de circuitos de niveles múltiples es valioso desde la teoría, pero en la práctica se hace difícil obtener circuitos que distingan en forma confiable entre más de dos valores. Por esta razón, el uso de la lógica de múltiples niveles no ha tenido gran desarrollo.

Este texto se ocupa fundamentalmente de circuitos digitales binarios, en los que las entradas y salidas pueden adoptar nada más que uno de dos valores. En consecuencia, solo se considerarán en el mismo señales digitales binarias.

A.3 Tablas de verdad

En el año 1854, George Boole publicó su trabajo de tesis sobre el uso de un álgebra que permitiese la representación de la lógica. El interés de Boole pasaba por capturar la matemática del pensamiento, para lo cual desarrolló una representación para información fáctica del tipo "la puerta está abierta" o "la puerta no está abierta". Posteriormente el álgebra de Boole fue desarrollada por Shannon, quien le dio la forma con la que hoy se conoce. En el álgebra de Boole se supone la existencia de un postulado básico, referido a una variable binaria que adopta el valor de cero o uno. Estos valores se condicen con los valores de 0 y 5 Volt mencionados en la sección anterior. La asignación puede invertirse en términos de las tensiones asignadas al 0 y al 1. Con el objeto de entender el funcionamiento de los circuitos digitales, puede omitirse la correspondencia física entre los valores lógicos y las tensiones, y considerar solo los valores simbólicos 0 y 1.

Una contribución fundamental de Boole es el desarrollo del concepto de **tabla de verdad**, la que captura e identifica relaciones lógicas en forma tabular. Considérese una habitación con dos llaves de tres vías *A* y *B* que actúan sobre una lámpara *Z*. Cada uno de los interruptores puede estar en cualquiera de sus dos posiciones. Cuando solo una de las dos llaves interruptoras está en su posición superior, la lámpara está encendida. Cuando las dos llaves están simultáneamente arriba o abajo, la lámpara está apagada. Se puede construir una tabla de verdad que detalle todas las posibles combinaciones de las posiciones de las llaves, según se muestra en la figura A.2. En la tabla, se le asigna el valor 0 a una llave cuando está en su posición inferior y, en caso contrario, se le asigna el valor 1. La lámpara está encendida cuando $Z = 1$.



Entradas		Salida
A	*B*	*Z*
0	0	0
0	1	1
1	0	1
1	1	0

Figura A.2 • Una tabla de verdad relaciona los estados de las llaves *A* y *B* con la lámpara *Z*.

Entradas		Salida
A	*B*	*Z*
0	0	1
0	1	0
1	0	0
1	1	1

Figura A.3 • Una asignación alternativa de la salida en función del estado de las llaves.

En una tabla de verdad, se enumeran todas las posibles combinaciones de entradas de las variables binarias y se asigna el correspondiente valor de 0 o 1 a la salida para cada combinación de entrada. En la tabla de verdad de la figura A.2, la salida *Z* depende de las variables de entrada *A* y *B*. Para cada combinación de las variables de entrada existen dos valores que pueden asignarse a *Z*: 0 o 1. Puede plantearse una asignación lógica diferente a la que se muestra en la figura A.2, en la que la lámpara se encienda solo cuando las dos llaves estén en la misma posición, es decir, ambas arriba o ambas abajo simultáneamente, caso en el cual la tabla de verdad de la figura A.3 detalla todos los posibles estados de la lámpara para cada combinación de las entradas. Cabe acotar que para cumplir con esta configuración debería modificarse el cableado eléctrico de las llaves. Para dos

variables de entrada existen cuatro combinaciones de entradas y, por ende, existen 2^4 posibles asignaciones de salidas para las cuatro combinaciones de entrada. En general, dado que n entradas permiten obtener 2^n combinaciones, existen $2^{(2^n)}$ posibles asignaciones de los valores de salida a las combinaciones de entrada.

A.4 Compuertas lógicas

Si se enumeran todas las posibles asignaciones de las posiciones de las llaves correspondientes a las dos variables de entrada, se obtendrán las 16 asignaciones que se muestran en la figura A.4. Estas funciones se conocen como **funciones lógicas booleanas**. Algunas de las asignaciones tienen nombres especiales. La función lógica *Y* (AND) es cierta (produce un 1) solo cuando *A* y *B* son 1, mientras que la función lógica *O* (OR) es cierta cuando *A* o *B* o ambas son ciertas. Una función es falsa cuando su salida es 0, por lo tanto, se denominará función *falsedad* a aquella cuyo valor siempre es 0, en tanto se denomina *certeza* a una función que es siempre cierta. En el álgebra de Boole, el signo de suma "+" representa una función lógica *O* y no implica una suma aritmética. Asimismo, la yuxtaposición de dos variables, tal como *A.B*, denota una función lógica *Y* entre las variables lógicas *A* y *B*.

Entradas		Salida							
A	*B*	*Falso*	*Y (AND)*	$A\overline{B}$	*A*	$\overline{A}B$	*B*	*XOR*	*O (OR)*
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Entradas		Salida							
A	*B*	*NOR*	*XNOR*	$\overline{B}$	$A+\overline{B}$	$\overline{A}$	$\overline{A}+B$	*NAND*	*Cierto*
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Figura A.4 • Tablas de verdad que ilustran todas las posibles funciones de dos variables binarias

Las funciones *A* y *B* simplemente repiten las entradas *A* y *B*, respectivamente, en tanto que las funciones $\overline{A}$ y $\overline{B}$ **complementan** a *A* y *B*, produciendo un 0 cuando la variable no complementada es 1 y un 1 cuando la variable no complementada es 0. En general, una barra encima de un término denota una operación de complemento, por consiguiente, las

funciones *NAND* y *NOR* son complementarias a *Y* (AND) y *O* (OR), respectivamente. La función *XOR* es cierta cuando cualquiera de sus entradas es cierta pero no cuando ambas son simultáneamente ciertas. La función *XNOR* es el complemento de la función *XOR*. Las restantes funciones se interpretan en forma similar.

Una **compuerta lógica** es un dispositivo físico que implementa una función básica del álgebra de Boole. Las funciones detalladas en la figura A.4 tienen representaciones simbólicas como compuertas lógicas, algunas de las cuales se muestran en las figuras A.5 y A.6. Para cada una de las funciones, *A* y *B* son las entradas binarias y *F* es la función.

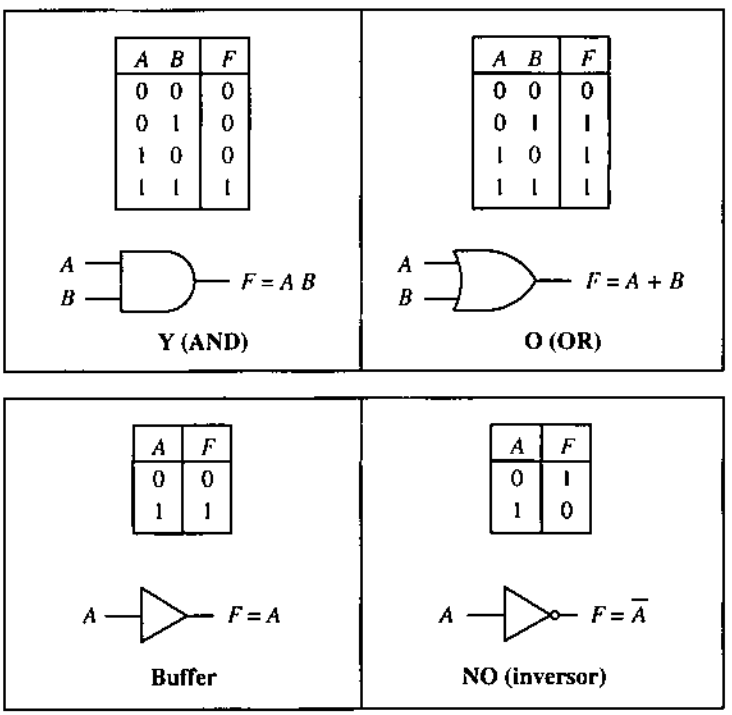


Figura A.5 • Símbolos lógicos de las funciones booleanas Y, O, *buffer* y NO.

La figura A.5 presenta las compuertas *O* e *Y*, cuyo comportamiento ya fue descripto. La salida de la compuerta Y es cierta cuando las dos entradas son simultáneamente ciertas y es falsa en los demás casos. La salida de la compuerta *O* es cierta cuando alguna o ambas entradas son ciertas y es falsa en el caso restante. La compuerta conocida como *buffer* simplemente copia su entrada sobre su salida. Si bien no tiene significación lógica alguna, juega un papel importante en la práctica cuando se la utiliza como amplificador, permitiendo que una cantidad de entradas de compuertas sean manejadas por la misma señal. La compuerta *NO* (conocida como **inversor**) produce un 1 en su salida cuando su entrada presenta un 0 y produce un 0 cuando la entrada está en 1. Nuevamente, se dice que la salida invertida corresponde al complemento de la entrada. El pequeño círculo ubicado a la salida de la compuerta NO indica la operación de complemento.

En la figura A.6, las compuertas *NAND* y *NOR* producen salidas complementarias a las de las salidas *Y* e *O*, respectivamente. La compuerta O excluyente (XOR) produce un 1 cuando alguna de sus salidas, pero no ambas, es 1. En general, la función *XOR* produce un 1 en su salida cuando el número de unos en sus entradas es impar. Esta generalización es importante para entender el comportamiento de una compuerta XOR con más de dos entradas. La compuerta *NOR* excluyente (*XNOR*) produce una salida complementaria a la de la *XOR*.

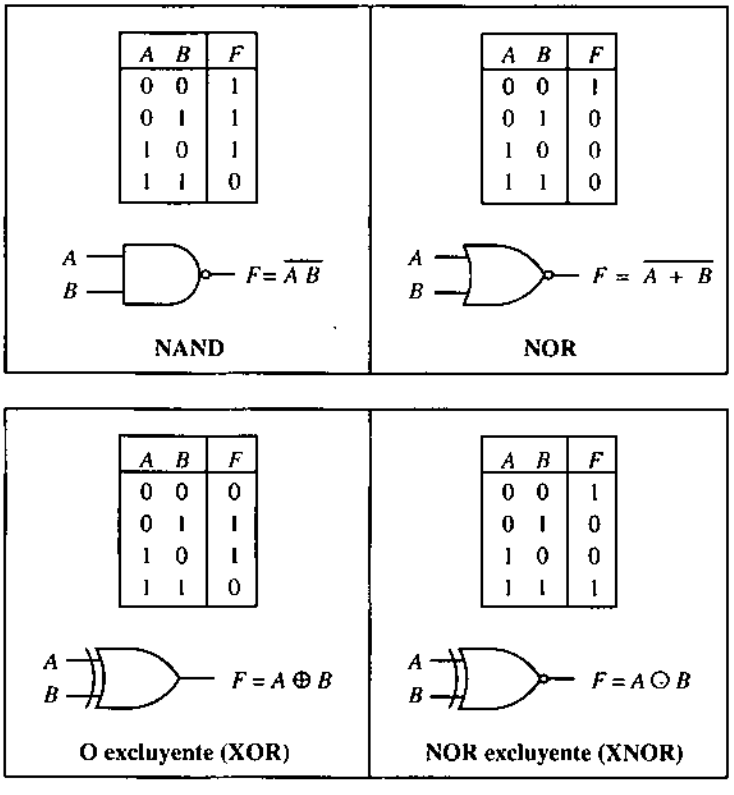


Figura A.6 • Símbolos de las compuertas lógicas representativas de las funciones booleanas NAND, NOR, XOR y XNOR.

Los símbolos lógicos de las figuras A.5 y A.6 solo indican las formas básicas, de las cuales existen y se usan a menudo distintas variantes. Por ejemplo, las compuertas pueden tener mayor número de entradas, como lo muestra la compuerta Y de tres entradas que se ilustra en la figura A.7a. Los círculos en las entradas de las compuertas *NO*, *NOR* y *XNOR* indican una operación de complemento y pueden ubicarse en las entradas de los elementos lógicos para indicar que las entradas se invierten al ingresar en la compuerta, tal como lo indica la figura A.7b. El símbolo lógico correspondiente a una compuerta que produce dos salidas complementarias se muestra en la figura A.7c.

Físicamente, las compuertas lógicas no son mágicas, aunque pueden verse así cuando un elemento como un inversor puede generar un 1 lógico (+5 V) en su salida cuando se

le coloca un 0 lógico (0 V) en su entrada. La próxima sección describe los mecanismos subyacentes que permiten el funcionamiento de las compuertas electrónicas.

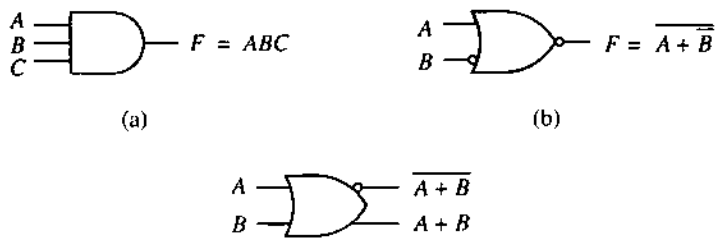


Figura A.7 • Variantes de los símbolos lógicos básicos para (a) tres entradas; (b) salida negada; (c) salidas complementarias.

A.4.1 Implementación electrónica de la lógica

Eléctricamente, las compuertas lógicas tienen terminales de alimentación que normalmente no se indican. La figura A.8a ilustra un inversor en el que se han indicado sus terminales de alimentación eléctrica de +5 V y 0 V (GND). La señal de +5 V se denomina V_{CC}, por "tensión de colector". En un circuito lógico, todos los terminales de V_{CC} y GND se conectan a los bornes correspondientes de una fuente de alimentación.

Las compuertas lógicas están constituidas por dispositivos eléctricos llamados **transistores**, los que tienen una característica básica de conmutación que les permite controlar una señal más fuerte con una señal más débil. Esta capacidad de amplificación es esencial para poder encadenar compuertas lógicas. Sin amplificación, solo se podría enviar una señal a través de unas pocas compuertas antes de que la señal se deteriore tanto que sea superada por niveles de ruido, el que existe, en grado mayor o menor, en cualquier punto de un circuito eléctrico.

La figura A.8b ilustra el símbolo esquemático de un transistor. Si no hay tensión positiva en la base del mismo, no habrá circulación de corriente entre V_{CC} y GND. Por lo tanto, para un inversor, un 0 lógico en la base (0 V) generará un 1 lógico (+5 V) en el terminal de colector, tal como se indica en la figura A.8c. Si, en cambio, se aplica una tensión positiva en V_{in}, habrá circulación de corriente entre V_{CC} y GND. Esta corriente hará que V_{out} no pueda producir la señal suficiente como para que la salida del inversor sea un 1. En efecto, si se aplican +5 V a la entrada V_{in}, aparece un 0 lógico en V_{out}. La relación de transferencia entre entrada y salida de una compuerta lógica sigue una curva no lineal, como la de la figura A.8d, que corresponde a un circuito lógico de la familia TTL (*transistor-transistor logic*). La alinealidad es una propiedad importante que hace posible la operación de compuertas en cascada.

Un paradigma útil es aquel que identifica la corriente que circula por un circuito eléctrico con el agua que fluye por una cañería. Si en una cañería se abre la conexión entre V_{CC} y GND, el agua que circula hacia V_{out} se reduce a un mínimo, si bien probablemen-

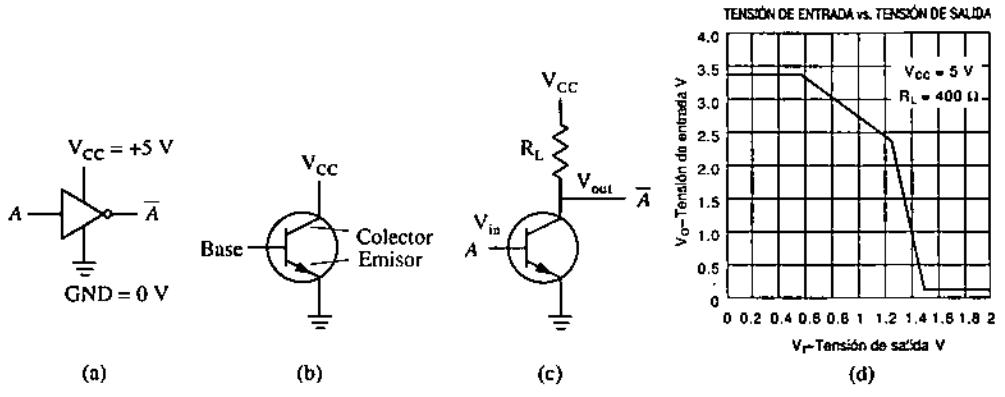


Figura A.8 • (a) Circuito inversor en el que se indican los terminales de alimentación; (b) símbolo esquemático de un transistor; (c) el circuito transistorizado de un inversor; (d) curva de transferencia estática de un inversor.

te haya alguna pequeña circulación de agua. Este efecto puede minimizarse eligiendo un valor adecuado para el resistor R_L.

Dado que siempre habrá algún flujo de corriente que circule aun teniendo un 0 lógico en V_{out}, se requiere asignar los valores lógicos de 0 y de 1 con un cierto margen de seguridad en las tensiones asignadas. Si se asignan los valores de 0 y 1 a 0 V y +5 V, respectivamente, puede ocurrir que el circuito no funcione en forma correcta si, por ejemplo, la salida de un inversor fuese de 0,1 V en lugar de ser estrictamente 0 V, como es habitual. Por esta razón, en el diseño de los circuitos lógicos, las asignaciones de los valores lógicos del 0 y del 1 se hacen mediante **umbrales**. En la figura A.9a, el 0 lógico se asigna al rango de tensiones que va desde 0 V hasta 0,4 V, en tanto que el 1 lógico se asigna al rango que va desde 2,4 V hasta +5 V. Los rangos de la figura A.9a corresponden a la salida de una compuerta lógica. Entre la salida de una compuerta lógica y la entrada de otra puede aparecer algún tipo de atenuación (reducción en los niveles de tensión) y, por ende, estos umbrales se corren 0,4 V en las entradas a las compuertas, como lo indica la figura A.9b. Estos rangos difieren según la familia lógica de que se trate.

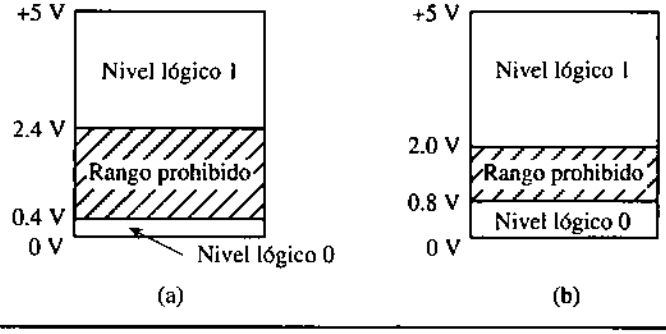


Figura A.9 • Asignación de los valores lógicos de 0 y 1 a los rangos de tensión (a) de salida de una compuerta; (b) de entrada de una compuerta lógica.

Los rangos de salida solo tienen sentido si las señales presentes en las entradas de la compuerta encajan dentro de los rangos correspondientes al 0 o al 1 lógico. Por esta razón, las entradas de una compuerta lógica nunca pueden quedar "flotando", es decir, desconectadas de otra salida, de V_{CC}, o de GND.

La figura A.10 ilustra circuitos transistorizados para las compuertas *NAND* y *NOR* de dos entradas. En el caso de la compuerta NAND, ambas entradas V_1 y V_2 deben estar en la zona de tensiones correspondientes al 1 lógico para producir en la salida V_{out} una tensión que se encuentre en el rango del 0 lógico. Para el caso de la compuerta NOR, si una o ambas entradas V_1 y V_2 están en la zona correspondiente al 1 lógico, V_{out} entregará un nivel ubicado en el rango correspondiente al 0 lógico.

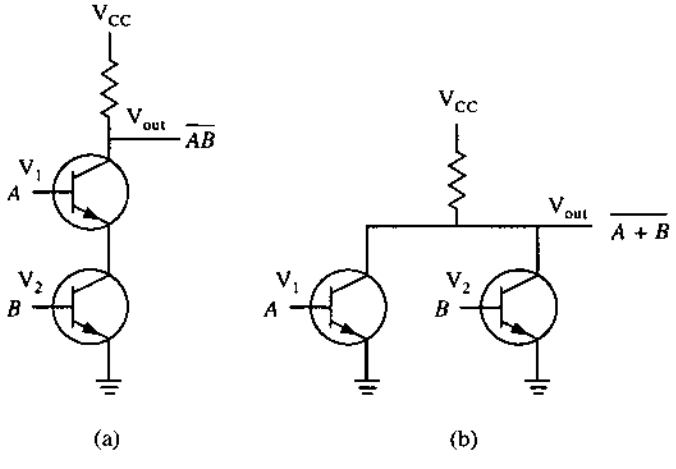


Figura A.10 • Circuitos transistorizados de (a) una compuerta NAND, (b) una compuerta NOR, ambas de dos entradas.

A.4.2 Buffers de tres estados

Un circuito ***buffer* de tres estados** funciona en forma similar a la de un circuito *buffer* convencional, planteado en una sección anterior de este apéndice, con el agregado de una entrada de control que permite deshabilitar el circuito. Según el valor de esta entrada de control, la salida del circuito vale 0, 1 o está desactivada, generando así tres estados de salida diferentes. En la figura A.11, cuando la entrada de control *C* vale 1, el circuito se comporta como un *buffer* convencional. Cuando *C* vale 0, en cambio, la salida queda eléctricamente desconectada, por lo que no se produce salida alguna. Las x de la tabla de verdad correspondiente indican los estados en los que el circuito se halla inhibido (desconectado). Debe hacerse notar que el estado de deshabilitación, x, no representa ni un 0 ni un 1, sino la ausencia de señal. En términos apropiados para un circuito eléctrico, la salida se encuentra en un estado de **alta impedancia**. El *buffer* de salida de tres estados con entrada de control invertida funciona en forma similar a la descripta, ex-

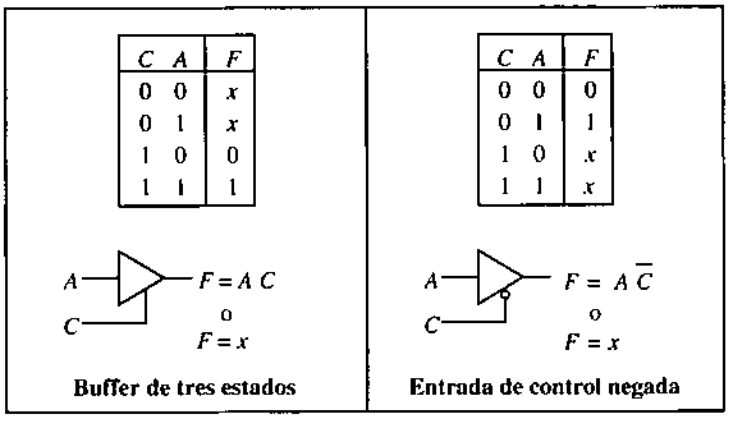


Figura A.11 • Circuitos *buffer* con salida de tres estados.

cepto por la inversión de la entrada de control C, la que se indica con el círculo en dicha entrada de control.

Una salida eléctricamente desconectada difiere de una salida que produce un 0, dado que la salida no conectada se comporta como si no hubiese conexión física en tanto que una salida en 0 sí está conectada al circuito siguiente. El *buffer* de tres estados permite que las salidas de distintas compuertas lógicas controlen una línea común sin riesgo de cortocircuitos eléctricos, dado que los mismos se habilitan de a uno por vez. El uso de estos *buffers* de tres estados se hace importante en la implementación de **registros** como los que se describirán en secciones posteriores de este apéndice.

A.5 Propiedades del álgebra de Boole

La tabla A.1 resume algunas de las propiedades básicas del álgebra de Boole que pueden aplicarse a las expresiones lógicas. Los postulados (conocidos como postulados de Huntington) son definiciones básicas del álgebra booleana, que por ser postulados no necesitan demostración alguna. Los teoremas se demuestran a partir de los postulados. Cada una de las relaciones indicadas en la tabla presenta dos formas, una para la suma y otra para el producto lógico, derivadas del llamado **principio de dualidad**. La forma dual de una expresión se obtiene intercambiando sumas lógicas (O) por productos (Y) y productos por sumas.

La propiedad **conmutativa** indica que el orden en el que aparezcan dos variables en las funciones suma y producto no es significativo. Por el principio de dualidad, la propiedad conmutativa tiene tanto una expresión para la suma lógica ($A + B = B + A$) como para el producto lógico ($A.B = B.A$). La propiedad **distributiva** indica la forma en que puede distribuirse una variable a través de un producto lógico que la incluya. Por aplicación del principio de dualidad, se obtiene la expresión dual para la suma lógica.

El principio de **identidad** indica que la suma lógica de una variable con un valor lógico de 0 o el producto de la misma con 1 dan por resultado la misma variable. La pro-

piedad de **complemento** indica que el producto lógico de una variable con su complemento da un resultado lógicamente falso (produce un 0 dado que al menos una de las entradas es 0), y que la suma lógica de una variable con su complemento es lógicamente cierta (produce un 1 dado que al menos una de las variables es 1).

Los teoremas del **cero** y del **uno** indican que el producto lógico de una variable con 0 siempre da 0, y que la suma lógica de una variable con 1 da siempre 1. El teorema de **idempotencia** dice que una variable afectada por una suma o producto lógico consigo misma reproduce el valor de la variable original. Por ejemplo, si las entradas de una compuerta *Y* o las entradas de una compuerta *O* tienen el mismo valor, la salida de cada una de las compuertas tendrá el mismo valor que las entradas. El teorema **asociativo** indica que el orden en que se realicen la suma y el producto lógicos no tienen importancia. El teorema de **involución** dice que el complemento del complemento de una variable o expresión reproduce la expresión o variable original.

	Relación	Dual	Propiedad
Postulados	$A\,B = B\,A$	$A + B = B + A$	Conmutativa
	$A\,(B + C) = A\,B + A\,C$	$A + B\,C = (A + B)\,(A + C)$	Distributiva
	$1\,A = A$	$0 + A = A$	Identidad
	$A\,\overline{A} = 0$	$A + \overline{A} = 1$	Complemento
Teoremas	$0\,A = 0$	$1 + A = 1$	Teoremas del cero y el uno
	$A\,A = A$	$A + A = A$	Idempotencia
	$A\,(B\,C) = (A\,B)\,C$	$A + (B + C) = (A + B) + C$	Asociativa
	$\overline{\overline{A}} = A$		Involución
	$\overline{A\,B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A}\,\overline{B}$	Teorema de DeMorgan
	$AB + \overline{A}C + BC = AB + \overline{A}C$	$(A + B)(\overline{A} + C)(B + C) = (A + B)(\overline{A} + C)$	Teorema del consenso
	$A\,(A + B) = A$	$A + A\,B = A$	Teorema de absorción

Tabla A.1 • Propiedades básicas del álgebra de Boole.

Los teoremas de **DeMorgan**, del **consenso** y de **absorción** pueden no resultar obvios. Se incluye una demostración del teorema de DeMorgan para un caso de dos variables utilizando un principio de inducción (enumerando todos los casos), dejando la demostración algebraica, así como una prueba inductiva del teorema del consenso, como ejercicios (véanse los problemas A.24 y A.25). La figura A.12 muestra una tabla de verdad para cada una de las expresiones que aparecen en las distintas formas del teorema de DeMorgan. Las expresiones que aparecen a derecha e izquierda en cada forma de los teoremas de DeMorgan generan resultados equivalentes, lo que prueba el teorema para el caso de dos variables.

A B	$\overline{AB} = \overline{A} + \overline{B}$		$\overline{A + B} = \overline{A}\,\overline{B}$	
0 0	1	1	1	1
0 1	1	1	0	0
1 0	1	1	0	0
1 1	0	0	0	0

Figura A.12 • Verificación del teorema de DeMorgan para dos variables.

No todas las compuertas lógicas descriptas hasta el momento son necesarias para lograr **estructuras lógicas completas**, lo que significa la posibilidad de crear cualquier circuito lógico a partir de estas compuertas. Existen tres conjuntos de compuertas lógicas que permiten lograr una estructura completa. Si bien existen otras alternativas, estos son los conjuntos formados por las compuertas *Y*, *O* y *NO*, por una parte, las compuertas *NAND* por otra y las compuertas *NOR* por otra.

Teorema de DeMorgan $A + B = \overline{\overline{A + B}} = \overline{\overline{A}\,\overline{B}}$

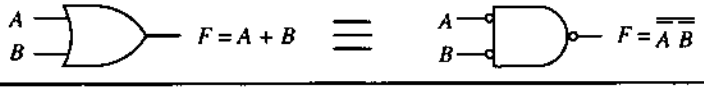


Figura A.13 • Uso del teorema de DeMorgan para generar una compuerta O con una compuerta NAND.

Un conjunto completo de compuertas permite generar otras compuertas lógicas que no son parte de dicho conjunto. Considérese como ejemplo la implementación de una función lógica *O* con el conjunto de compuertas *NAND*. De acuerdo con lo que se ilustra en la figura A.13, se puede utilizar el teorema de DeMorgan para convertir una compuerta *O* en una compuerta *NAND*. La función original $(A + B)$ se complementa dos veces, lo que, según la propiedad de involución, la mantiene intacta. El teorema de DeMorgan reemplaza las operaciones *O* por *Y*, además de distribuir la barra de complemento interior que afecta a las variables *A* y *B*. Por aplicación de la propiedad de idempotencia, las entradas invertidas pueden implementarse también como compuertas *NAND*, como se indica en la figura A.14. Así, la función *O* se implementa únicamente con compuertas *NAND*. La equivalencia funcional entre compuertas lógicas es importante en lo que hace a la aplicación práctica, dado que un tipo de compuerta lógica puede tener mejores características operativas que otro, dependiendo de la tecnología utilizada.

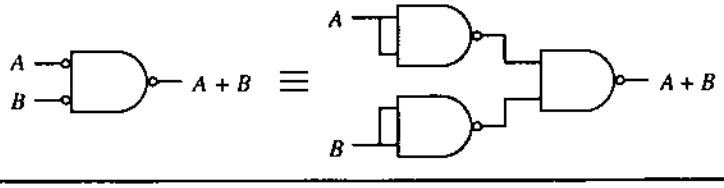


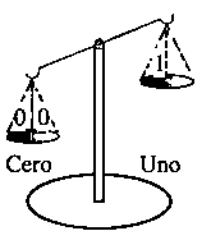
Figura A.14 • Las entradas invertidas de una compuerta NAND se implementan con compuertas NAND.

A.6 Representación en suma de productos y diagramas lógicos

Supóngase ahora la necesidad de implementar una función más compleja que una simple compuerta lógica, tal como la **función mayoría** descripta por la tabla de verdad de la figura A.15. La función mayoría es cierta cada vez que más de la mitad de sus variables lógicas de entrada son ciertas, y puede imaginarse como una balanza que se inclina a izquierda o a derecha dependiendo de si hay más ceros o unos en las entradas. Este tipo de operación suele ser común para la recuperación de errores, en la que se comparan las salidas de circuitos idénticos que operan sobre los mismos datos, a partir de lo cual se determina la salida real por medio del mayor número de salidas similares (algo así como una votación).

Dado que ninguna de las compuertas lógicas analizadas hasta el momento puede ejecutar en forma directa la función planteada, se la transforma en una ecuación lógica de dos niveles *Y-O*, tras lo cual se la implementa con compuertas lógicas, usando, por ejemplo, el conjunto *Y*, *O*, *NO*. El concepto de lógica de dos niveles se debe a que en la implementación se obtiene un único nivel de variables afectadas por una operación de producto lógico *Y*, seguido por otro nivel lógico correspondiente a una operación *O*. La ecuación booleana que describe la función mayoría es cierta cada vez que *F* es cierta en la tabla de verdad. Así, *F* es cierta cuando $A = 0$, $B = 1$ y $C = 1$, o cuando $A = 1$, $B = 0$ y $C = 1$, y así sucesivamente en los restantes casos.

Término mínimo / *Índice*	*A*	*B*	*C*	*F*
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1



La balanza se inclina en uno u otro sentido dependiendo de si hay más ceros o unos.

Figura A.15 • Tabla de verdad correspondiente a la función mayoría.

Una forma de representar ecuaciones lógicas requiere el uso de la forma **suma de productos** (SOP, *sum of products*), en la cual se realiza una suma lógica (O) de un conjunto de productos lógicos (Y) entre las distintas variables. La expresión lógica que describe la función mayoría en formato suma de productos se indica en la ecuación A.1. Como ya se ha dicho, el símbolo "+" representa la suma lógica y en ningún caso implica operación alguna de suma aritmética.

$$F = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC \qquad \text{(A.1)}$$

Al inspeccionar la ecuación, surge que para implementar los cuatro **términos producto** $\bar{A}BC$, $A\bar{B}C$, $AB\bar{C}$ y ABC se requieren cuatro compuertas Y de tres entradas cada una, cuyas

salidas podrán conectarse a las entradas de una compuerta O de cuatro entradas, como lo ilustra la figura A.16. Este circuito resuelve la función mayoría, lo que puede verificarse al enumerar cada una de las ocho combinaciones de las entradas y analizar la salida en cada caso.

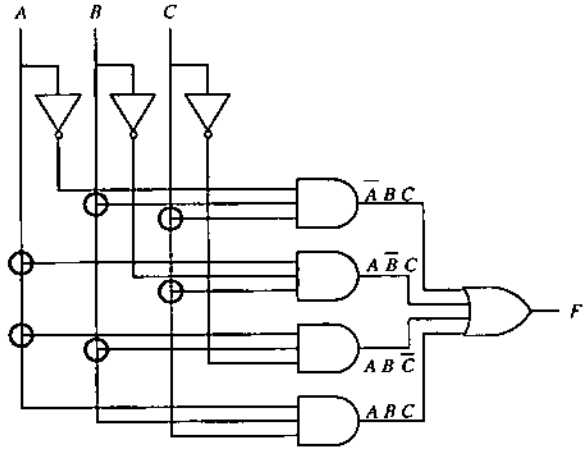


Figura A.16 • Implementación circuital de la función mayoría utilizando dos niveles Y-O. Los inversores de las entradas no se incluyen al calcular los dos niveles.

El diagrama circuital muestra una notación habitualmente utilizada para representar la presencia o ausencia de conexión eléctrica, que se resume en la figura A.17. Dos líneas que se cruzan no se conectan, a menos que en el punto de intersección se coloque un símbolo de conexión, normalmente indicado con un punto oscuro. Dos líneas que se encuentran en forma de T se conectan tal como se indica en las intersecciones resaltadas, por lo que en esos puntos de intersección no se requiere colocar símbolo de conexión.

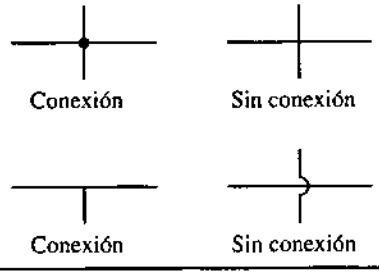


Figura A.17 • Cuatro convenciones utilizadas para indicar intersecciones.

Cuando un producto lógico contiene todas las variables de la función, expresadas una sola vez, sea en su forma real o complementada, recibe la denominación de **término mínimo**. Un término mínimo toma el valor 1 para una única entrada de la tabla de verdad. Esto significa que la función será cierta para un mínimo número (uno) de términos. Como alternativa, la función suele escribirse como la suma lógica de las entradas ciertas de la tabla. La ecuación A.1 podría reescribirse como lo indica la ecuación A.2, en la que los

índices utilizados corresponden a los nombres asignados a los términos mínimos en la columna izquierda de la figura A.15.

$$F = \sum(3, 5, 6, 7) \tag{A.2}$$

Esta notación es la apropiada para representar la forma **canónica** de una ecuación booleana, la que solo contiene términos mínimos. Las ecuaciones A.1 y A.2 representan la función en su "forma canónica suma de productos".

A.7 La forma producto de sumas

Como expresión dual a la forma suma de productos, una expresión booleana puede representarse en una forma **producto de sumas** (POS, *product of sums*). Una ecuación representada en su forma producto de sumas contiene conjuntos de variables afectadas por operaciones de suma lógica, cuyos resultados se convierten en un producto lógico. Un modo de obtener una forma producto de sumas nace de complementar la expresión suma de productos y aplicar los teoremas de DeMorgan. Por ejemplo, volviendo a la tabla de verdad de la función mayoría de la figura A.15, se obtiene el complemento de la función seleccionando los términos de entrada que hacen 0 la salida, lo que se expresa en la ecuación A.3.

$$\overline{F} = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\overline{C} \tag{A.3}$$

De complementar ambos miembros de la igualdad, surge la ecuación A.4:

$$F = \overline{\overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}BC + A\overline{B}\overline{C}} \tag{A.4}$$

La aplicación del teorema de DeMorgan en la primera negación, en la forma siguiente $\overline{W + X + Y + Z} = \overline{W}\overline{X}\overline{Y}\overline{Z}$, genera la ecuación A.5:

$$F = (\overline{\overline{A}\overline{B}\overline{C}})(\overline{\overline{A}\overline{B}C})(\overline{\overline{A}B\overline{C}})(\overline{A\overline{B}\overline{C}}) \tag{A.5}$$

Una nueva aplicación del teorema de DeMorgan a los términos entre paréntesis, ahora en su forma $\overline{WXYZ} = \overline{W} + \overline{X} + \overline{Y} + \overline{Z}$, , da origen a la ecuación A.6:

$$F = (A + B + C)(A + B + \overline{C})(A + \overline{B} + C)(\overline{A} + B + C) \tag{A.6}$$

La ecuación A.6 está expresada en la forma producto de sumas y contiene cuatro **términos máximos**, en los que cada variable de la función aparece una sola vez en forma real o complementada. Un término máximo, tal como $(A + B + C)$, adopta un valor 0 para una

única entrada de la tabla de verdad. Esto es, es cierto para el máximo número de entradas de la tabla de verdad que se obtiene antes de reducir la función a su situación trivial de ser siempre cierta. Una ecuación que contiene solo términos máximos en la forma producto de sumas se encuentra expresada en su forma canónica producto de sumas. El circuito O-Y que implementa la ecuación A.6 es el indicado en la figura A.18. La forma O-Y es lógicamente equivalente a la forma Y-O de la figura A.16.

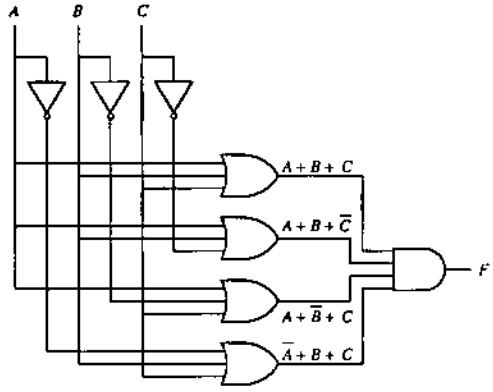


Figura A.18 • La función mayoría implementada con dos niveles Y-O. Nuevamente, no se incluyen los inversores en la cuenta de los dos niveles.

Un argumento para el uso de la forma producto de sumas en reemplazo de la forma suma de productos puede ser la obtención de una expresión booleana más sencilla. Una expresión booleana más sencilla, a su vez, puede significar un circuito más simple, aun cuando esto no siempre es válido debido a que existen distintas consideraciones que no dependen directamente del tamaño de la ecuación booleana, como puede ser la complejidad de la topología de cableado.

La **cantidad de compuertas** es una muestra de la complejidad del circuito, que surge de contar la totalidad de compuertas que forman el circuito. La **cantidad de entradas de compuertas** es otra forma de medir la complejidad de un circuito. Esta se obtiene contando el total de entradas existente en el total de las compuertas que forman el circuito. Para los circuitos de las figuras A.16 y A.18, ambas formas, suma de producto y producto de sumas, dan un total de ocho compuertas y un total de 19 entradas. Por consiguiente, no hay diferencia en la complejidad de ambas formas circuitales, aun cuando en otros casos las diferencias puedieran ser significativas. Existen distintos métodos para reducir la complejidad de los circuitos digitales, algunos de los cuales se presentan en el apéndice B.

A.8 Lógica positiva o lógica negativa

Hasta este punto se ha planteado la suposición de que los valores de tensión alto y bajo se corresponden con los niveles lógicos de 1 y 0, o de CIERTO y FALSO, respectivamente, lo que se suele conocer como **activo alto** o **lógica positiva**. Puede plantearse la asignación in-

versa: el menor valor de tensión para el 1 lógico y el mayor valor de tensión para el 0 lógico. Esta última convención se conoce como **activo bajo** o **lógica negativa**. En ciertos casos se prefiere el uso de lógica negativa, en lugar de la lógica positiva, especialmente cuando se trata de aplicaciones en las cuales la lógica inhibe un evento en lugar de habilitarlo.

La figura A.19 ilustra el comportamiento de los pares de compuertas Y-O y O-Y tanto para lógica positiva como negativa. La compuerta de lógica positiva Y se comporta como una compuerta O en lógica negativa. La compuerta física es la misma, independientemente de la convención positiva o negativa de la lógica. Lo único que cambia es la implementación de las señales.

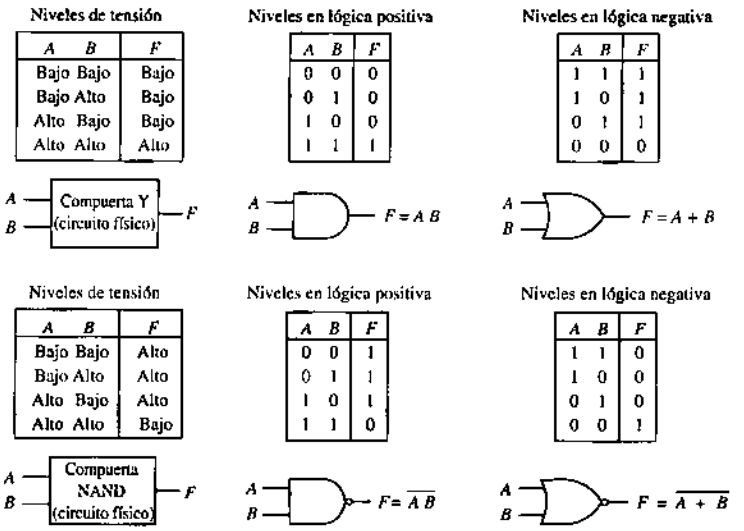


Figura A.19 • Asignaciones de lógica positiva y negativa para pares Y-O y O-Y.

Con el objeto de no generar confusiones, debería evitarse la mezcla de lógica positiva y negativa en el mismo sistema, aunque en ciertos casos esta mezcla no puede evitarse. Por ejemplo, con el objeto de mantener el sentido lógico siempre correcto, se suele usar una técnica conocida como "de burbujas coincidentes". La idea es la de suponer que toda la lógica es positiva y colocar una burbuja o círculo de inversión a la entrada de cualquier circuito que funcione en lógica negativa. Nótese que estas burbujas cumplen con la misma función que los círculos que se colocan en las salidas negadas de circuitos lógicos, como las compuertas NOR y NAND. Esto es, la señal que sale de un círculo es el complemento de la que ingresa al mismo.

Si se considera el circuito de la figura A.20a, se observa que las salidas de dos circuitos de lógica positiva se encuentran combinados a través de una compuerta Y que, a su vez, se conecta a otro sistema de lógica positiva. La figura A.20b ilustra un circuito lógicamente equivalente pero en lógica negativa. En el proceso de coincidencia, se coloca un círculo inversor en cada entrada o salida de lógica negativa, como lo indica la figura A.20c.

Con el objeto de simplificar el análisis del circuito, las negaciones de las entradas activas en bajo deben adecuarse a las negaciones de las salidas activas en bajo. En la figura

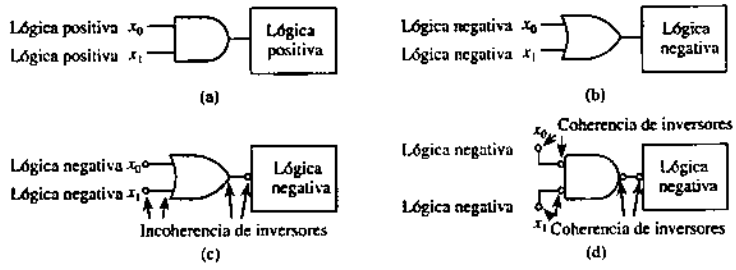


Figura A.20 • El método para hacer coincidir las burbujas.

A.20c hay fallas de coincidencia debido a que hay una sola negación en cada línea. Se utiliza el teorema de DeMorgan para convertir el circuito O de la figura A.20a en el circuito de compuertas NAND con entradas negadas planteado en la figura A.20d, en el cual se han solucionado las fallas de coincidencia del circuito anterior.

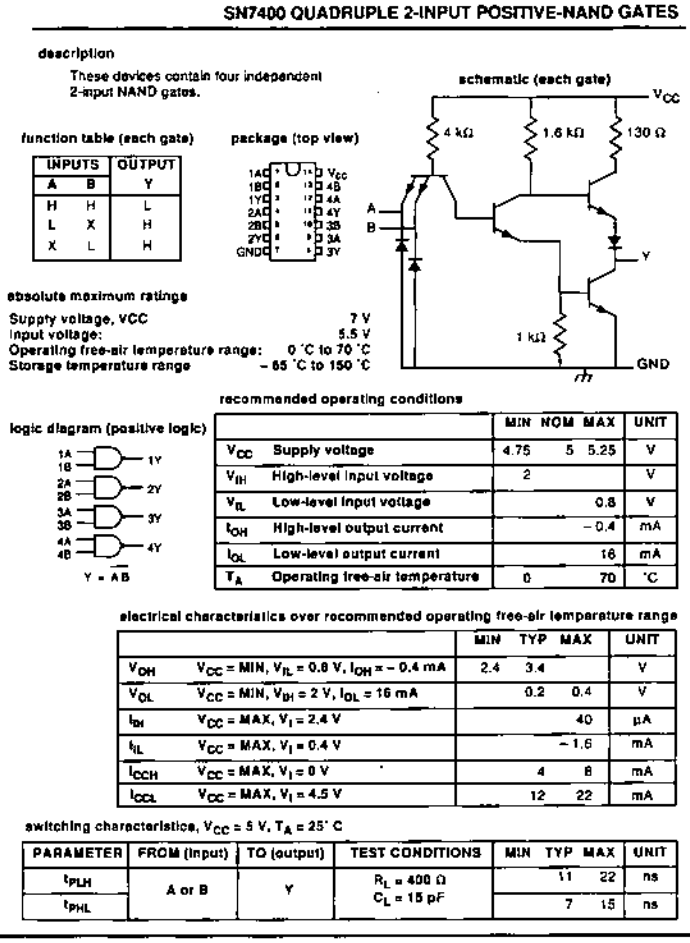
A.9 La hoja de datos

Las compuertas lógicas y los restantes componentes circuitales tienen una gran cantidad de especificaciones técnicas que resultan relevantes al momento del diseño y análisis de los circuitos digitales. La **hoja de datos**, también llamada hoja de especificaciones, de un circuito detalla las características técnicas de un componente lógico. La figura A.21 muestra un ejemplo de una hoja de datos. La misma se inicia con la identificación del componente, que en este caso corresponde a la compuerta lógica NAND SN7400. A continuación se incluye un texto con la descripción funcional del componente.

Una segunda sección (*package*) muestra la distribución y asignación de terminales del circuito integrado. Pueden existir distintos tipos de encapsulado para el mismo componente. La tabla funcional detalla, desde el punto de vista funcional, el comportamiento de entradas y salidas del componente. Los símbolos "H" y "L" se utilizan para representar, respectivamente, los valores alto (*high*) y bajo (*low*) de tensión, con lo que se evitan confusiones respecto de si la lógica usada es positiva o negativa. El símbolo "X" se utiliza para indicar que el valor de una entrada no afecta al de una salida. El diagrama lógico describe el comportamiento lógico del componente, utilizando en este caso lógica positiva. Se indican las cuatro compuertas NAND que forman el circuito con sus asignaciones de terminales de entrada y salida.

El esquema circuital muestra cada compuerta a su nivel de transistores. En este texto, el circuito de bajo nivel se trata como una abstracción que está incluida en los símbolos lógicos de la compuerta.

La sección correspondiente a los máximos valores admisibles ("*absolute maximum ratings*") indica el entorno de condiciones ambientales en las que el componente opera en forma segura. La alimentación eléctrica puede llegar hasta los 7 V, en tanto que las tensiones de entrada pueden llegar hasta 5,5 V. La temperatura ambiente durante el fun-



SN7400 QUADRUPLE 2-INPUT POSITIVE-NAND GATES

description

These devices contain four independent 2-input NAND gates.

function table (each gate)

INPUTS A	INPUTS B	OUTPUT Y
H	H	L
L	X	H
X	L	H

package (top view)

1A 1 — 14 V_{CC}
1B 2 — 13 4B
1Y 3 — 12 4A
2A 4 — 11 4Y
2B 5 — 10 3B
2Y 6 — 9 3A
GND 7 — 8 3Y

schematic (each gate)

V_{CC}
4 kΩ
1.6 kΩ
130 Ω
A
B
Y
1 kΩ
GND

absolute maximum ratings

Supply voltage, VCC: 7 V
Input voltage: 5.5 V
Operating free-air temperature range: 0 °C to 70 °C
Storage temperature range: − 65 °C to 150 °C

logic diagram (positive logic)

1A, 1B — 1Y
2A, 2B — 2Y
3A, 3B — 3Y
4A, 4B — 4Y
$Y = \overline{AB}$

recommended operating conditions

		MIN	NOM	MAX	UNIT
V_{CC}	Supply voltage	4.75	5	5.25	V
V_{IH}	High-level input voltage	2			V
V_{IL}	Low-level input voltage			0.8	V
I_{OH}	High-level output current			− 0.4	mA
I_{OL}	Low-level output current			16	mA
T_A	Operating free-air temperature	0		70	°C

electrical characteristics over recommended operating free-air temperature range

		MIN	TYP	MAX	UNIT
V_{OH}	V_{CC} = MIN, V_{IL} = 0.8 V, I_{OH} = − 0.4 mA	2.4	3.4		V
V_{OL}	V_{CC} = MIN, V_{IH} = 2 V, I_{OL} = 16 mA		0.2	0.4	V
I_{IH}	V_{CC} = MAX, V_I = 2.4 V			40	μA
I_{IL}	V_{CC} = MAX, V_I = 0.4 V			− 1.6	mA
I_{CCH}	V_{CC} = MAX, V_I = 0 V		4	8	mA
I_{CCL}	V_{CC} = MAX, V_I = 4.5 V		12	22	mA

switching characteristics, V_{CC} = 5 V, T_A = 25° C

PARAMETER	FROM (input)	TO (output)	TEST CONDITIONS	MIN	TYP	MAX	UNIT
t_{PLH}	A or B	Y	R_L = 400 Ω, C_L = 15 pF		11	22	ns
t_{PHL}					7	15	ns

Figura A.21 • Hoja de datos simplificada para la compuerta NAND 7400 (adaptado del TTL Databook de Texas Instruments).

cionamiento del circuito debería mantenerse entre 0°C y 70°C, pero puede variar entre -65°C y 150°C cuando el componente no se utiliza.

A pesar de las especificaciones máximas absolutas establecidas, durante la operación del circuito se deben respetar las condiciones recomendadas de operación. Estas condiciones recomendadas se caracterizan por valores mínimos (MIN), nominales (NOM) y máximos (MAX).

Las características eléctricas describen la conducta del componente ante ciertas condiciones de operación. V_{OH} y V_{OL} son los valores mínimo y máximo, respectivamente, que puede adoptar una salida en sus estados alto y bajo. I_{IH} e I_{IL} representan la máxima corriente en un terminal de entrada cuando esa entrada está, respectivamente, en sus estados alto y bajo. I_{CCH} e I_{CCL} corresponden a la corriente tomada por el circuito integrado de la fuente de alimentación cuando todas las salidas están en estado alto y bajo, respectivamente.

Esta información puede usarse para determinar la **máxima capacidad de carga** (*fan out*) bajo las condiciones dadas. La capacidad de carga es una medida de la cantidad de entradas que puede manejar una salida dada, considerando compuertas implementadas en la misma tecnología. Esto significa que si en una compuerta se tiene un *fan out* de 10, la misma puede manejar las entradas de 10 compuertas del mismo tipo. En forma similar, el concepto de ***fan in*** determina una medida de la cantidad de entradas que una compuerta lógica puede aceptar (dicho simplemente, el número de entradas en esa compuerta).* En una salida dada, el valor absoluto de I_{OH} debe ser mayor o igual que la suma de todas las corrientes I_{IH} de las compuertas controladas, de la misma forma en que el valor absoluto de I_{OL} debe ser mayor o igual que la suma de todas las corrientes I_{IL} de las compuertas a las que la salida está conectada. El valor absoluto de I_{OH} para una compuerta 7400 es de 0,4 mA (o 400 μ), por lo que una compuerta 7400 puede manejar en su salida diez entradas TTL, cada una de las cuales requiere una corriente I_{IH} de 40 μ.

Las características de conmutación indican los tiempos de propagación requeridos para conmutar la salida desde el nivel bajo de tensión a su nivel alto (t_{PLH}) o desde su valor bajo a su nivel alto (t_{PHL}). Los valores máximos muestran los que corresponden al peor caso. El diseño de un circuito puede hacerse en forma segura tomando como valores de peor caso los valores típicos, pero únicamente si se utiliza un método de diseño que permita medir y luego elegir el mejor de los componentes utilizados. Dado que los valores de t_{PLH} pueden variar entre 11 ns y 22 ns, y los de t_{PHL}, a su vez, pueden variar entre 7 ns y 15 ns para componentes encapsulados en un mismo integrado, esto significa poder medir individualmente cada componente para determinar sus características reales. No todos los componentes del mismo tipo se comportan de la misma manera, aún bajo los métodos de fabricación más exigentes, pudiendo reducir las diferencias a partir de mediciones y la posterior selección de los mejores componentes.

A.10 Componentes digitales

En los diseños de circuitos digitales de alto nivel suelen utilizarse, además de las compuertas individuales, conjuntos de compuertas lógicas denominadas **componentes** o bloques funcionales. Esto permite abstraer un cierto nivel de complejidad circuital así como caracterizar su eficiencia. La próxima sección describe algunos de los componentes más comunes.

---

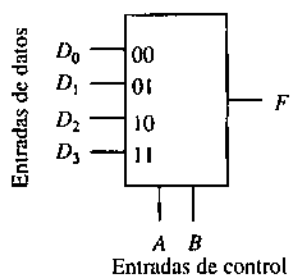
* *N. de T.*: Como un concepto más riguroso, se suele definir como *fan in* de una entrada a un circuito lógico, a la cantidad de entradas a las que dicha entrada equivale. Por ejemplo, si para una tecnología dada la corriente de entrada I_{IL} es de 1,6 mA, una entrada de un circuito que tenga por su estructura una corriente de entrada de 4,8 mA presenta un *fan in* de 3, dado que su corriente de entradas equivale a la corriente de entrada de tres compuertas básicas.

A.10.1 Niveles de integración

Hasta este punto se ha puesto énfasis en el diseño de circuitos lógicos combinatorios. Dado que el análisis fue realizado con compuertas lógicas individuales, se puede decir que el enfoque ha sido el de utilizar circuitos **integrados en pequeña escala** (SSI, *small scale integration*), en los que hay 10 a 100 elementos por circuito integrado. (La palabra "elementos", en este caso, hace referencia a transistores y otros elementos discretos.) Aun cuando en la práctica a veces se requiere trabajar en este nivel, típicamente para circuitos de alta eficiencia, el advenimiento de la microelectrónica permite trabajar en niveles de integración mayores. En la tecnología de **integración en media escala** (MSI, *medium scale integration*) aparecen entre 100 y 1.000 componentes en un único circuito. La tecnología de **integración en gran escala** (LSI, *large scale integration*) trata con circuitos que contienen 1.000 a 10.000 componentes por circuito integrado, y la tecnología de **integración en mayor escala** (VLSI, *very large scale integration*) va todavía más allá. No hay fronteras estrictas entre las distintas clases de integración, pero las distinciones son útiles para comparar la complejidad relativa de los distintos circuitos. El tratamiento de esta sección tiene que ver básicamente con componentes MSI.

A.10.2 Multiplexores

Un circuito **multiplexor** (MUX) es un elemento que conecta una cantidad dada de entradas a una salida única. En la figura A.22 se muestra el diagrama en bloques y la tabla de verdad de un multiplexor de 4 entradas y una salida. La salida F adopta el valor correspondiente a la entrada de datos seleccionada por las líneas de control A y B. Por ejemplo, si $AB = 00$, el valor que aparece en la salida es el que corresponde a la entrada D_0. El circuito Y-O correspondiente se muestra en la figura A.23.



A	B	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

$$F = \bar{A}\,\bar{B}\,D_0 + \bar{A}\,B\,D_1 + A\,\bar{B}\,D_2 + A\,B\,D_3$$

Figura A.22 • Diagrama en bloques y tabla de verdad para un multiplexor de 4 entradas de datos.

Cuando se diseñan circuitos utilizando multiplexores, se los considera como un bloque funcional, el que puede representarse como una "caja negra" según se ve en la figura A.22. Si se utiliza este criterio, en vez de considerar el circuito lógico de la figura A.23, se evitan los detalles innecesarios en el diseño de circuitos complejos.

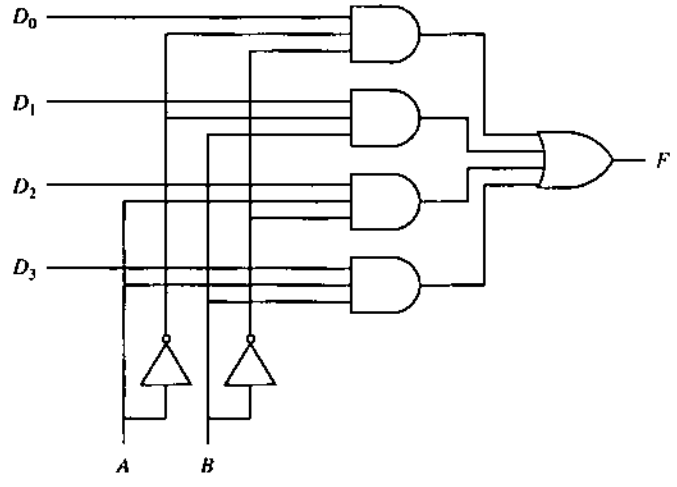


Figura A.23 • Circuito Y-O que implementa el multiplexor de cuatro entradas.

La implementación de funciones booleanas puede realizarse por medio de multiplexores. En la figura A.24 se implementa la función mayoría usando un multiplexor de 8 entradas. Las entradas de datos se toman directamente de la tabla de verdad de la función implementada, y se asignan las variables A, B y C como entradas de control. El multiplexor implementa la función transfiriendo a la salida los unos correspondientes a cada término mínimo de la función. Las entradas cuyos valores son 0 corresponden a los elementos del multiplexor que no se requieren para la implementación de la función, y como resultado hay compuertas lógicas que no se utilizan. Si bien en la implementación de funciones booleanas siempre hay porciones del multiplexor que no se utilizan, el uso de multiplexores es amplio debido a que su generalidad simplifica el proceso de diseño y su modularidad simplifica la implementación.

A	B	C	M
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

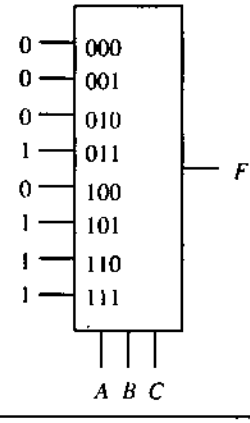


Figura A.24 • La función mayoría implementada con un multiplexor de 8 entradas.

Como otro ejemplo, considérese la implementación de una función de tres variables usando un multiplexor de cuatro entradas. La figura A.25 ilustra una tabla de verdad de

tres variables y un multiplexor de cuatro entradas que implementa la función F. Las entradas de datos se toman del conjunto $\{0, 1, C, \overline{C}\}$ y la agrupación se obtiene de acuerdo con lo que se muestra en la tabla de verdad. Cuando $AB = 00$, la función $F = 0$ independientemente del valor de C y, por lo tanto, la entrada de datos 00 del multiplexor tendrá un valor fijo de 0. Cuando $AB = 01$, $F = 1$ independientemente de C, por lo que la entrada de datos 01 adopta un valor 1. Cuando $AB = 10$, la función $F = C$ dado que su valor es 0 cuando C es 0 y es 1 cuando C es 1. Finalmente, cuando $AB = 11$, la función $F = \overline{C}$, por lo tanto, la entrada de datos 11 adopta el valor $\overline{C}$. De esta manera, se puede implementar una función de tres variables usando un multiplexor con dos entradas de control.

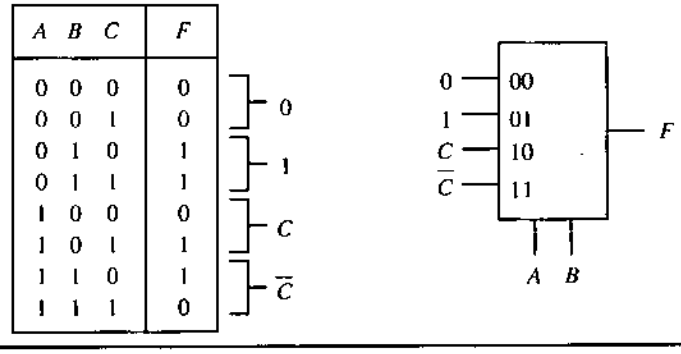
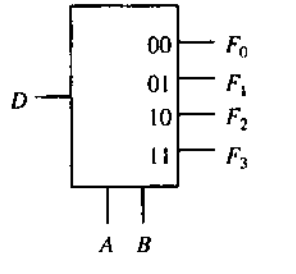


Figura A.25 • Una función de tres variables implementada con un multiplexor de cuatro entradas de datos.

A.10.3 Demultiplexores

Un **demultiplexor** (DEMUX) es un circuito que cumple la función inversa a la de un multiplexor. La figura A.26 ilustra el diagrama en bloques correspondiente a un demultiplexor de cuatro salidas, cuyas entradas de control son A y B, y su correspondiente tabla de verdad. Un demultiplexor envía su única entrada de datos D a una de sus salidas F_i, de acuerdo con los valores que adopten sus entradas de control. La figura A.27 muestra el circuito



$F_0 = D\overline{A}\,\overline{B}$ $\quad F_2 = DA\overline{B}$

$F_1 = D\overline{A}B$ $\quad F_3 = DAB$

D	A	B	F_0	F_1	F_2	F_3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Figura A.26 • Diagrama en bloques y tabla de verdad de un demultiplexor de cuatro salidas.

de un demultiplexor de cuatro salidas. Una de las aplicaciones en las que se utilizan los demultiplexores es el envío de datos desde un origen único hacia un conjunto de destinos, como en el caso de un circuito de llamada de ascensores que deriva la llamada al ascensor más cercano. Los demultiplexores no son de uso habitual en la implementación de funciones booleanas, aun cuando existen formas de aplicarlos (véase el problema A.17).

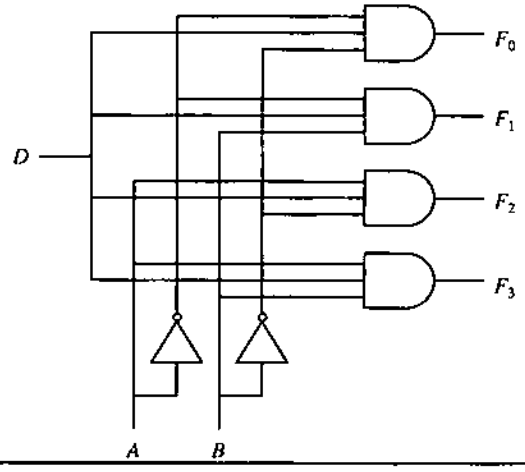
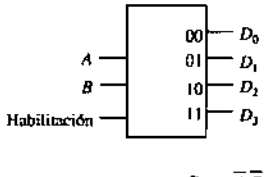


Figura A.27 • Circuito lógico de un demultiplexor de cuatro salidas.

A.10.4 Decodificadores

Un **decodificador** traduce una codificación lógica hacia una ubicación espacial. En cada momento, solo una de las salidas del decodificador está en el estado activo (1 lógico), según lo que determinen las entradas de control. La figura A.28 muestra el diagrama en bloques y la tabla de verdad de un decodificador de 2 entradas a 4 salidas, cuyas entradas de control son A y B. El diagrama lógico correspondiente a la implementación del decodificador se muestra en la figura A.29. Un circuito decodificador puede usarse para controlar otros circuitos, aunque a veces puede resultar inadecuado habilitar cualquiera de esos otros circuitos. Por esa razón, se incorpora en el circuito decodificador una línea de habilitación, la que fuerza todas las salidas a nivel 0 (inactivo) cuando se le aplica un 0 en la entrada. (Nótese la equivalencia lógica entre el demultiplexor con su entrada en 1 y el decodificador.)



Habilitación = 1						Habilitación = 0					
A	B	D_0	D_1	D_2	D_3	A	B	D_0	D_1	D_2	D_3
0	0	1	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	1	0	0	0	0
1	0	0	0	1	0	1	0	0	0	0	0
1	1	0	0	0	1	1	1	0	0	0	0

$D_0 = \overline{A}\,\overline{B}$ $D_1 = \overline{A}\,B$ $D_2 = A\,\overline{B}$ $D_3 = A\,B$

Figura A.28 • Diagrama en bloques y tabla de verdad de un decodificador 2 a 4.

Una aplicación para un circuito decodificador puede ser la traducción de direcciones de memoria a sus correspondientes ubicaciones físicas. También puede utilizarse en la implementación de funciones booleanas. Dado que cada línea de salida corresponde a un término mínimo distinto, puede implementarse una función por medio de la suma lógica de las salidas correspondientes a los términos que son ciertos en la función. Por ejemplo, en la figura A.30 se puede ver la implementación de la función mayoría con un decodificador de 3 a 8. Las salidas no utilizadas se dejan desconectadas.

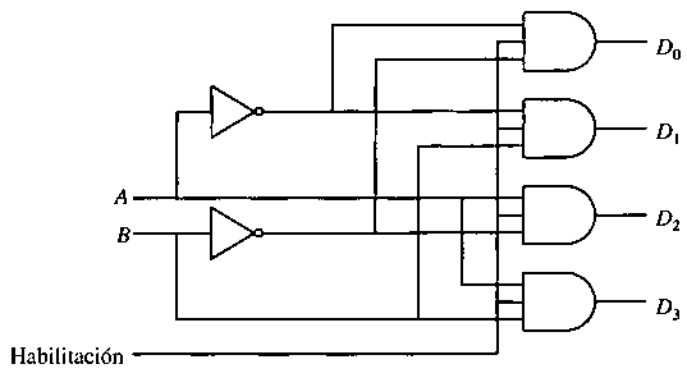


Figura A.29 • Un circuito de compuertas Y para el decodificador 2 a 4.

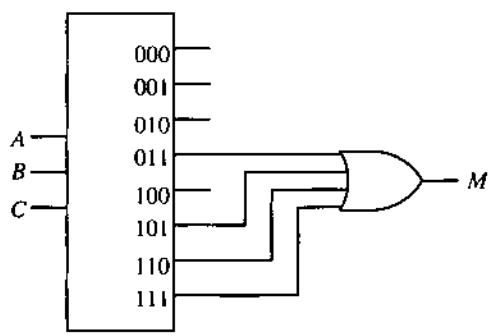
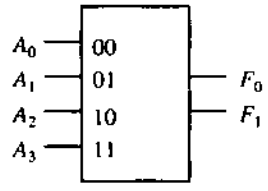


Figura A.30 • La función mayoría implementada por un decodificador 3 a 8.

A.10.5 Codificadores de prioridad

Un **codificador** traduce un conjunto de entradas en un código binario y puede pensarse como el circuito opuesto al de un decodificador. Un **codificador de prioridad** es un tipo de codificador en el que se establece un ordenamiento de las entradas. El diagrama en bloques y la tabla de verdad de un codificador de prioridad de 4 entradas a 2 salidas se muestra en la figura A.31. El esquema de prioridades impuesto sobre las entradas hace que A_i tenga una prioridad mayor que A_{i+1}. La salida de dos bits adopta los valores 00, 01, 10 u 11, dependiendo de las entradas activas y de sus prioridades relativas. Cuando no hay entradas activas, las salidas llevan, por defecto, a asignarle prioridad a la entrada A_0 ($F_0F_1 = 00$).



$$F_0 = \overline{A_0}\,\overline{A_1}A_3 + \overline{A_0}A_1A_2$$
$$F_1 = \overline{A_0}A_2A_3 + \overline{A_0}A_1$$

A_0	A_1	A_2	A_3	F_0	F_1
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	1	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	0	1
0	1	1	1	0	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	0	0
1	1	1	1	0	0

Figura A.31 • Diagrama en bloques y tabla de verdad de un codificador de prioridad de 4 a 2.

Los codificadores de prioridad se utilizan para arbitrar entre una cantidad de dispositivos que compiten por un mismo recurso, como cuando se produce el intento de acceso simultáneo de una cantidad de usuarios a un sistema de computación. La figura A.32 ilustra el diagrama circuital para un codificador de prioridad de 4 entradas y 2 salidas. (El circuito ha sido simplificado utilizando métodos descriptos en el apéndice B, pero su comportamiento de entrada-salida puede verificarse sin necesidad de conocer el método de reducción que se ha utilizado.)

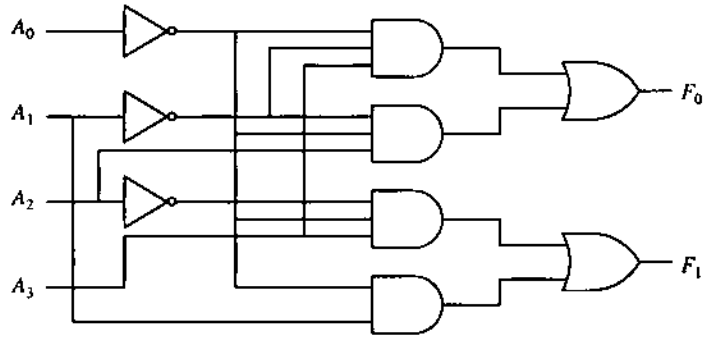


Figura A.32 • Diagrama lógico de un codificador de prioridad de 4 a 2.

A.10.6 Matrices lógicas programables

Una **matriz lógica programable** (PLA, *programmable logic array*) es un componente que consiste en una matriz configurable de compuertas Y seguida de otra matriz configurable de compuertas O. La figura A.33 muestra una matriz lógica programable de tres en-

tradas y dos salidas. Las tres entradas A, B, C y sus complementos se encuentran disponibles en las entradas de cada una de ocho compuertas Y que generan ocho términos en forma de producto lógico. Se dispone de las salidas de las compuertas Y en las entradas de cada una de las compuertas O que generan las funciones F_0 y F_1. En cada intersección de cada una de las dos matrices se coloca un fusible programable. Las matrices se personalizan para una función específica anulando fusibles. Cuando se inhabilita un fusible en la entrada de una compuerta Y, esa compuerta se comporta como si la entrada estuviese conectada a 1. En forma similar, una entrada inhabilitada en una compuerta O de una matriz lógica programable hace que se comporte como si esa entrada estuviese conectada a 0.

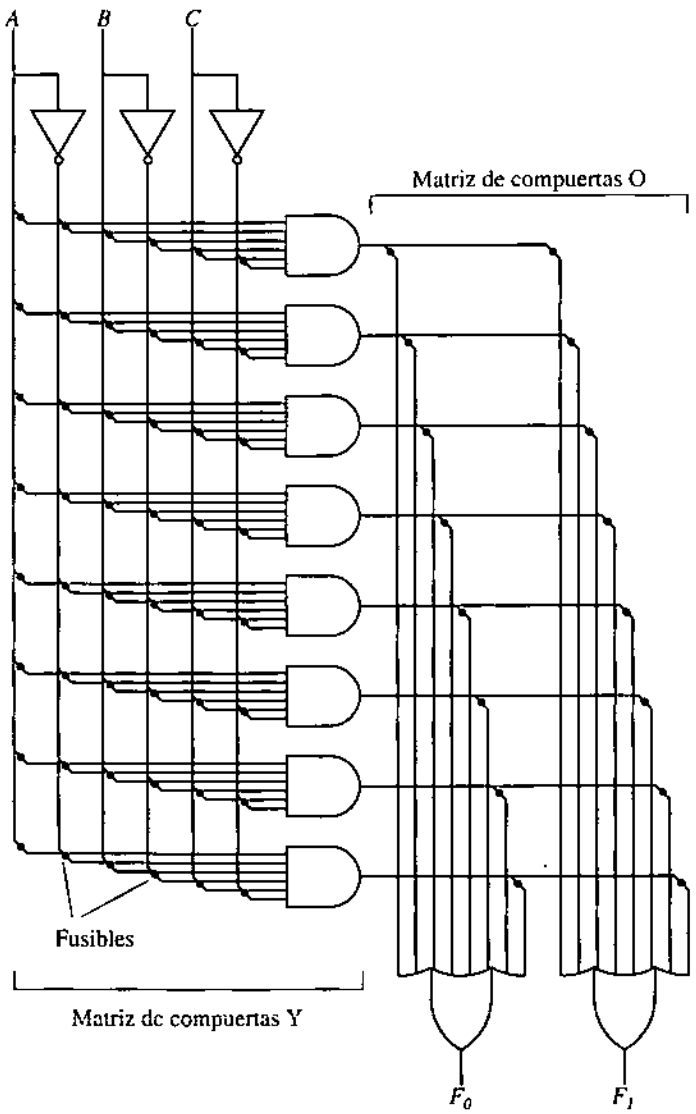


Figura A.33 • Una matriz lógica programable.

Como ejemplo de utilización de una matriz programable, considérese la implementación de la función mayoría en una matriz lógica programable de 3 x 2 (tres variables de entrada por dos funciones de salida). Con el objeto de simplificar las ilustraciones, se utiliza un esquema como el de la figura A.34, en el que se sobreentiende que la única línea de entrada a cada compuerta Y representa seis líneas de entrada, así como la única línea de entrada a cada compuerta O representa ocho líneas de entrada. Los puntos de conexión se colocan en los puntos de cruce para indicar las conexiones realizadas. En la figura A.34, la función mayoría se implementa utilizando solo la mitad de la matriz, lo que deja disponible la otra mitad para alguna otra función de las mismas variables.

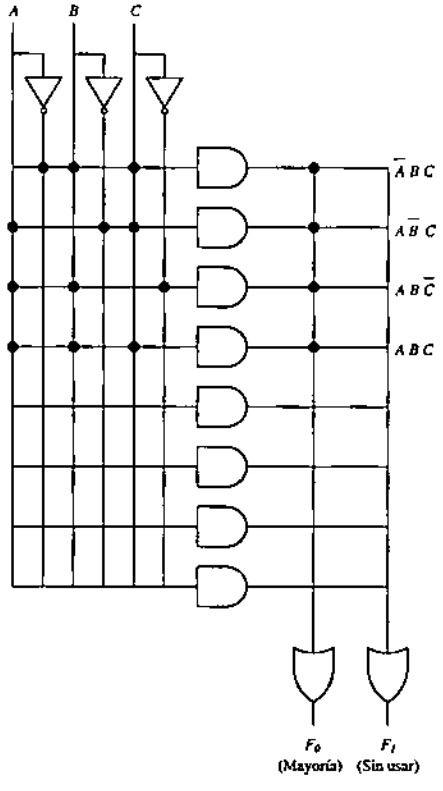


Figura A.34 • Representación simplificada de una matriz lógica programable.

Las matrices lógicas programables son componentes de gran aplicación en los sistemas digitales. La ventaja obtenida al utilizar estos elementos es que se tienen pocas entradas y pocas salidas, aun cuando haya un número importante de compuertas entre las entradas y las salidas. Es importante minimizar la cantidad de conexiones del circuito con el objeto de obtener un sistema modular formado por componentes discretos que se diseñan y se implementan por separado. Una matriz lógica programable es un elemento ideal en este caso; además, se dispone de buena cantidad de programas que diseñan matrices a partir de descripciones funcionales. Manteniendo el criterio modular, se suele representar una

matriz lógica programable como una caja negra del tipo de la que se indica en la figura A.35; en este caso, se supone que el diseño de la matriz puede dejarse en manos de un programa automatizado sin riesgo alguno.

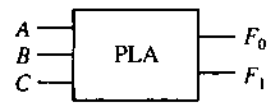


Figura A.35 • Representación de una matriz lógica programable como una caja negra.

Ejemplo: un sumador con arrastre serie

Como ejemplo de la utilización de matrices lógicas programables en el diseño de circuitos digitales, considérese el diseño de un circuito que suma dos números binarios. La suma binaria se realiza en forma similar a la que se utiliza para sumar manualmente dos números decimales, según se muestra en la figura A.36. Los dos números binarios *A* y *B* se suman de derecha a izquierda, creando en cada columna un bit de suma y uno de arrastre. En cada columna se tienen que sumar dos bits correspondientes a los operandos y un arrastre de entrada, por lo que en cada columna se deben tener en cuenta ocho posibles combinaciones, según lo muestra la tabla de verdad de la figura A.37.

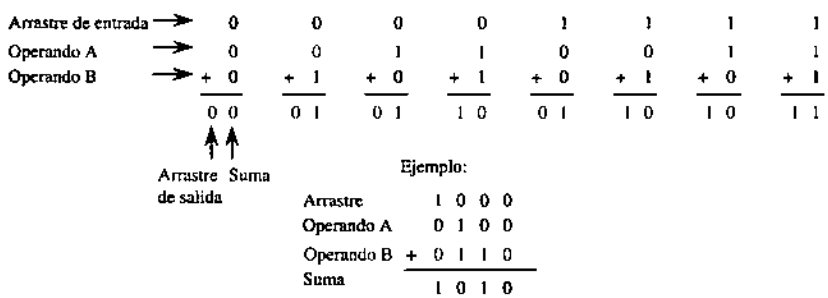


Figura A.36 • Ejemplo de suma de dos números binarios sin signo.

La tabla de verdad de la figura A.37 describe un elemento conocido como **sumador completo**, el que se muestra en forma esquemática en la figura. Un **semisumador**, que podría utilizarse en la columna de la derecha (la de las unidades) suma dos bits y genera una suma y un arrastre, en tanto que un sumador completo suma dos bits más uno de arrastre y genera un bit de suma y uno de arrastre. No se utiliza el semisumador en este ejemplo para lograr la mínima cantidad de componentes distintos. Se conectarán en cascada cuatro sumadores completos para formar un sumador lo suficientemente grande como para permitir la suma de los dos números usados en el ejemplo de la figura A.36. Este sumador se ilustra en la figura A.38, en la cual el sumador de la derecha recibe un arrastre de entrada siempre en 0.

A_i	B_i	C_i	S_i	C_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

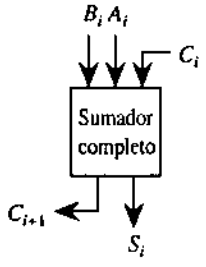


Figura A.37 • Tabla de verdad de un sumador completo.

El lector podrá apreciar que el valor de la suma en una columna dada no puede obtenerse hasta que se determine el valor del arrastre que sale de la columna anterior. El circuito se denomina sumador con arrastre en serie, debido a que los valores correctos del arrastre viajan a través del circuito, de derecha a izquierda. Asimismo, el lector puede observar que aun cuando el circuito tiene aspecto de funcionar en paralelo, en realidad los resultados de cada columna se determinan en secuencia empezando desde la derecha. Esta es una desventaja fundamental de este tipo de circuitos. En el capítulo 3 se analizan los métodos que permiten acelerar la suma.

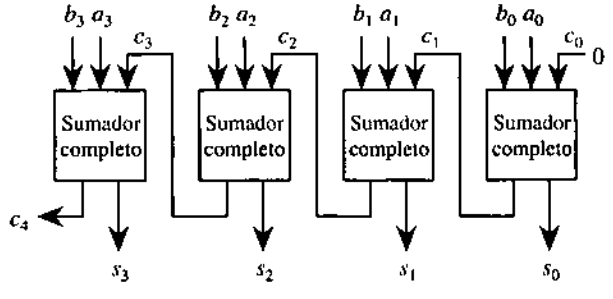


Figura A.38 • Sumador de cuatro bits implementado con sumadores completos conectados en cascada.

Una forma de diseñar un sumador completo consiste en la utilización de una matriz lógica programable, como se muestra en la figura A.39. El diseño mediante matrices lógicas programables es muy general, y las herramientas de diseño asistido por computadora (CAD, *computer aided design*) habitualmente favorecen el uso de matrices por sobre el uso de lógica discreta o de multiplexores, debido a su generalidad. Las herramientas de diseño asistido normalmente reducen los tamaños de las matrices lógicas (en el apéndice B se tratarán algunas de las técnicas de reducción), por lo que la gran cantidad de compuertas aparentemente utilizadas en una matriz lógica programable no es tan elevada en la práctica.

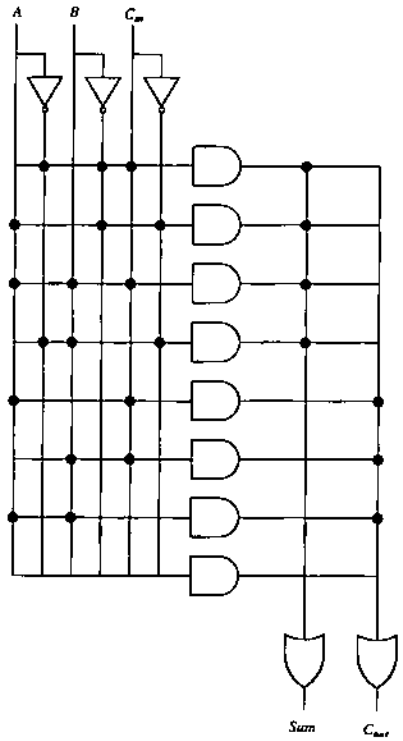


Figura A.39 • Implementación de un sumador completo con una matriz lógica programable.

A.11 Lógica secuencial

La primera parte de este apéndice fue dedicada a los circuitos lógicos combinatorios, en los que las salidas quedan totalmente determinadas como función de las entradas. Un circuito lógico secuencial, conocido como **máquina de estados finitos**, toma una entrada y un estado actual para generar una salida y un nuevo estado. Esta máquina de estados finitos se distingue de un circuito combinatorio en que la historia de las entradas del circuito secuencial tiene influencia sobre sus estados y sobre su salida. Este concepto es importante en la implementación de circuitos de memoria, así como en el diseño de unidades de control en una computadora.

El modelo clásico de una máquina de estados finitos se ilustra en la figura A.40. Consiste en un circuito combinatorio que toma entradas desde las líneas i_0-i_k, externas a la máquina de estados finitos, además de tomar entradas desde las líneas de estado s_0-s_n, internas a la máquina de estados. La unidad combinatoria genera los bits de salida f_0-f_m y un nuevo conjunto de bits de estado. Los elementos de retardo mantienen el estado actual de la máquina de estados finitos hasta que una señal de sincronismo provoque la carga de los valores D_i en los elementos S_i, tras lo que aparecen en Q_i como los nuevos bits de estado.

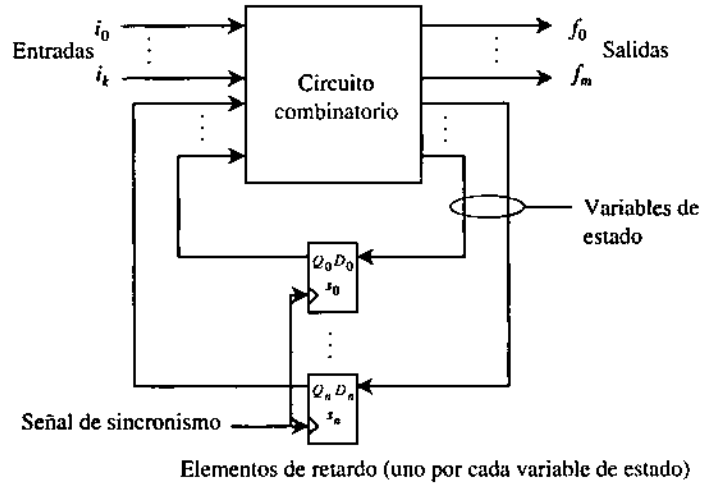


Figura A.40 • Modelo clásico de una máquina de estados finitos.

A.11.1 El circuito biestable (flip flop) S-R

Un **circuito biestable**, o *flip flop*, es un conjunto de compuertas lógicas que mantiene estable el estado de la salida aun luego de que las entradas pasen a un estado inactivo. La salida de un *flip flop* queda determinada tanto por las entradas actuales como por la historia de las mismas; en consecuencia, no es suficiente el uso de un circuito combinatorio para capturar este comportamiento. Un *flip flop* se puede utilizar para almacenar un bit de información, sirviendo además como bloque constructivo para memorias de computadora.

Si una o ambas entradas de una compuerta NOR de dos entradas está en 1, la salida de la compuerta NOR es 0; en los demás casos la salida vale 1. Según se ha visto con anterioridad en este apéndice, el tiempo requerido para que una señal se propague desde las entradas de una compuerta lógica hasta sus salidas no es nulo, y existe un cierto retardo $\Delta\tau$, que representa el tiempo de propagación a través de la compuerta. Para el análisis, el retardo suele considerarse acumulado a la salida de la compuerta, como se indica en la figura A.41. El retardo acumulado no se indica en los diagramas circuitales pero su presencia está implícita.

El tiempo de propagación a través de la compuerta NOR afecta el funcionamiento de un *flip flop*. Considérese el *flip flop* **set-reset** (S-R) de la figura A.42, que consiste en dos compuertas NOR conectadas entre sí. Si se aplica un 1 en la entrada S, la salida $\overline{Q}$ pasa a adoptar el valor 0 luego de un tiempo $\Delta\tau$, lo que hace que la salida Q adopte el valor 1 (suponiendo 0 como estado inicial de R) luego de un retardo $2\ \Delta\tau$. Como resultado de este tiempo de propagación finito (no nulo), durante un pequeño instante de tiempo $\Delta\tau$ las dos salidas Q y $\overline{Q}$ adoptan el valor 0, lo que no corresponde desde el punto de vista lógico. Esta situación se solucionará en el análisis posterior de la configuración circuital conocida como **maestro-esclavo** (*master-slave*). Si se aplica ahora un 0 en la entra-

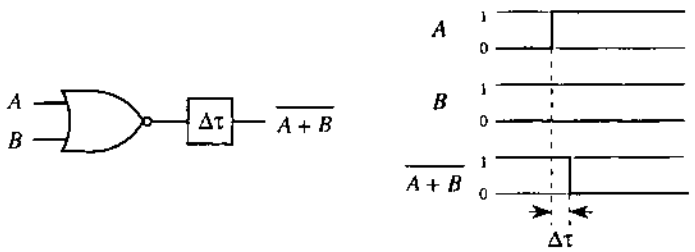
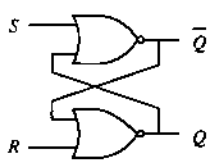


Figura A.41 • Una compuerta NOR con un retardo acumulado en la salida.

da S, la salida Q mantendrá su estado hasta algún momento posterior en que se lleve la entrada R a 1. El *flip flop* S-R, por ende, retiene un único bit de información y sirve como elemento básico de memoria.



Q_t	S_t	R_t	Q_{t+1}
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	(Prohibido)
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	(Prohibido)

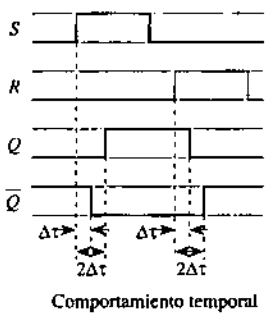


Figura A.42 • Un circuito *flip flop* S-R.

Existe más de una manera de plantear un *flip flop* S-R, por lo que el uso de compuertas NOR interconectadas es solo una de esas configuraciones. Puede implementarse un *flip flop* S-R utilizando dos compuertas NAND interconectadas, caso en el cual el estado de reposo es el que corresponde a $S = R = 1$. Con el uso del teorema de DeMorgan, se pueden convertir las compuertas NOR de un *flip flop* S-R en compuertas Y, según se ve en la figura A.43. Operando con inversores, se reemplazan las compuertas Y por compuertas NAND, y luego se invierten los sentidos activos de S y de R para eliminar los inversores de entrada remanentes.

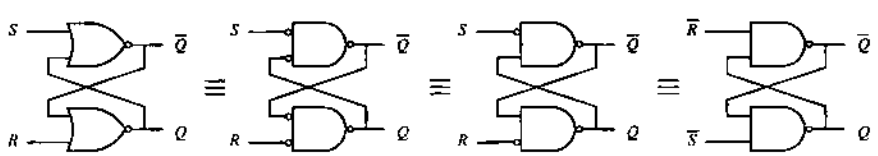


Figura A.43 • Conversión de un *flip flop* S-R implementado con compuertas NOR en una implementación con compuertas NAND.

A.11.2 El flip flop S-R sincrónico

Considérese ahora que las entradas del *flip flop* S-R pueden generarse desde las salidas de algún otro circuito cuyas entradas, a su vez, pueden generarse desde las salidas de otros circuitos, formando una cascada de circuitos lógicos. Esta estructura refleja el formato de los circuitos digitales convencionales. No obstante, en la interconexión en cascada de circuitos lógicos pueden surgir problemas con aquellas transiciones que se produzcan en momentos indeseados.

Considérese el circuito de la figura A.44. Si las señales *A*, *B* y *C* cambian todas desde su estado lógico 0 al estado 1, la señal *C* puede llegar a la compuerta XOR antes de que *A* y *B* hayan completado su propagación a través de la compuerta Y. Esto producirá un 1 momentáneo en la salida correspondiente a la entrada *S*, salida que volverá a 0 cuando la salida de la compuerta Y se termine de establecer y se realice la operación de XOR correspondiente. En este punto, puede ocurrir que *S* haya estado en 1 el tiempo suficiente como para llevar a 1 la salida del *flip flop*, destruyendo eventualmente la integridad del bit almacenado.

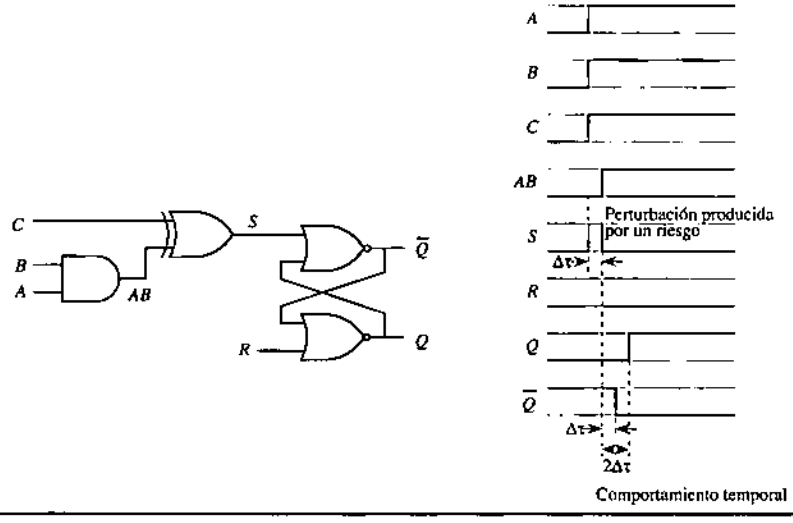


Figura A.44 • Un circuito con riesgos.

Cuando en un circuito biestable sus estados son sensibles a la secuencia temporal en que recibe las señales de entrada, la consecuencia puede ser una **variación transitoria** de algún estado o salida del circuito, los que, por indeseados, derivan en un mal funcionamiento del mismo.* Se suele decir que los circuitos que pueden producir perturbaciones transitorias indeseadas tienen **riesgos**. Un riesgo puede manifestarse como una perturbación en las salidas o no, según las condiciones operativas del circuito en un momento dado.

* *N. de T.*: El término "*glitch*", utilizado en el original, suele emplearse sin traducción en la jerga electrónica para nombrar este tipo de perturbaciones transitorias producidas en una salida o en un estado de un circuito lógico.

Con el objeto de lograr una sincronización controlada de los circuitos de lógica secuencial, se suele incorporar una señal de **sincronismo** o **reloj**, con la cual cada circuito de estados (como lo son los *flip flops*) se sincroniza a sí mismo al aceptar cambios de sus entradas solo en instantes determinados. Un circuito de reloj produce una secuencia continua de ceros y unos, como lo ilustra la forma de onda de la figura A.45. El tiempo que requiere la señal de reloj para subir, caer y volver a subir se conoce como **período** de la señal. Los flancos rectos que se ilustran en la forma de onda de la figura representan una señal cuadrada ideal. En la práctica, los flancos no son verticales debido a que el crecimiento y la caída de la señal no son instantáneos.

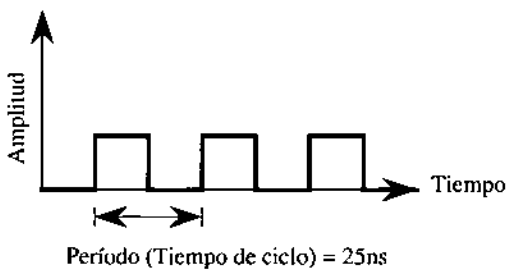


Figura A.45 • Forma de onda de una señal de reloj.

Se define como **frecuencia** de la señal de reloj a la inversa de su período. Para un período de 25 ns/ciclo, la frecuencia correspondiente es de 1/25 ciclos/ns, lo que corresponde a 40 millones de ciclos por segundo, o 40 MHz. La tabla A.2 muestra una serie de abreviaturas que se utilizan habitualmente en la representación de períodos y frecuencias.

Prefijo	Abreviatura	Cantidad	Prefijo	Abreviatura	Cantidad
Mili	m	10^{-3}	Kilo	K	10^{3}
micro	μ	10^{-6}	Mega	M	10^{6}
nano	n	10^{-9}	Giga	G	10^{9}
pico	p	10^{-12}	Tera	T	10^{12}
femto	f	10^{-15}	Peta	P	10^{15}
atto	a	10^{-18}	Exa	E	10^{18}

Tabla A.2 • Prefijos normalizados para la denominación de períodos y frecuencias.

El uso de una señal de sincronismo permite la eliminación de riesgos por medio de la creación de un **circuito biestable sincrónico**, que se muestra en la figura A.46, el que incluye una entrada *CLK* como señal de sincronismo. Las entradas *S* y *R* ya no pueden cambiar el estado del circuito hasta que no se reciba un nivel alto en *CLK*. Por consiguiente, si los cambios en *S* y *R* se producen mientras la señal de reloj está en su estado inactivo (bajo), cuando la señal de reloj pase a 1 los nuevos estados de *S* y *R*, estables, se almacenarán en el *flip flop*.

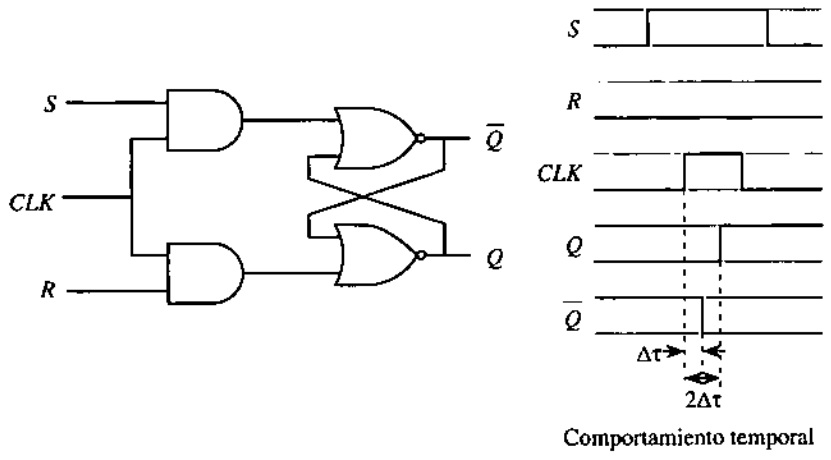


Figura A.46 • Un circuito S-R sincrónico.

A.11.3 El flip flop D y la configuración maestro-esclavo

El circuito S-R analizado tiene una desventaja: para almacenar un 1 o un 0, hace falta aplicar un 1 a alguna de dos entradas diferentes (*S* o *R*) dependiendo del valor que se pretenda almacenar. Una alternativa diferente, que permite aplicar un 0 o un 1 a la misma entrada, lleva a la configuración correspondiente a un ***flip flop*** D, que se ilustra en la figura A.47. El *flip flop* D se obtiene conectando las entradas *S* y *R* entre ellas a través de un circuito inversor. Cuando se activa la señal de reloj, el valor de *D* queda almacenado en el *flip flop*.

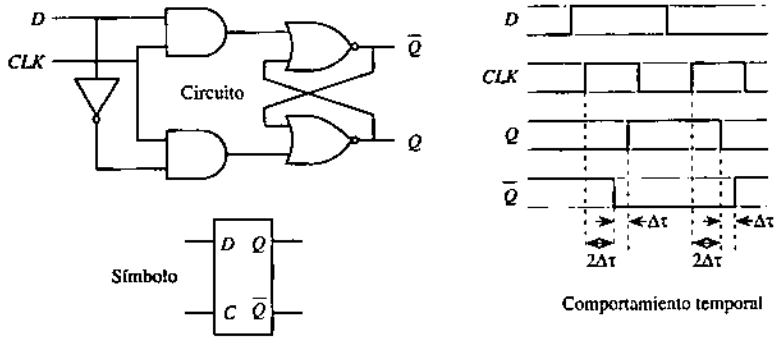


Figura A.47 • Un circuito D sincrónico. La entrada C representa la señal de sincronismo en la forma simbólica del *flip flop*.

Cuando se usa el *flip flop* D en situaciones en las que existe realimentación desde la salida hacia la entrada a través de otros circuitos, esta realimentación puede provocar que el *flip flop* cambie sus estados más de una vez en un ciclo de reloj. Con el objeto de asegurar que el *flip flop* cambia una sola vez por ciclo de reloj, se suele cortar el lazo de realimentación a través de la estructura conocida como **maestro-esclavo**, que se muestra en la fi-

gura A.48. El *flip flop* maestro-esclavo consiste en dos *flip flops* encadenados, donde el segundo utiliza una señal de sincronismo que está negada con respecto a la que utiliza el primero de ellos. El *flip flop* maestro cambia cuando la entrada principal de reloj está en su estado alto, pero el esclavo no puede cambiar hasta que esa entrada no vuelva a bajar. Esto significa que la entrada D se transfiere a la salida Q_S del *flip flop* esclavo recién cuando la señal de reloj sube y vuelve a bajar. El triángulo utilizado en el símbolo del *flip flop* maestro-esclavo indica que las transiciones de la salida ocurren solo en un flanco creciente (transición 0-1) o decreciente (transición 1-0) de la señal de reloj. No se producen transiciones continuas en la salida cuando la señal de reloj se encuentra en su nivel alto, como ocurre con el circuito sincrónico simple. Para la configuración de la figura A.48, la transición de la salida se produce en el flanco negativo de la señal de sincronismo.

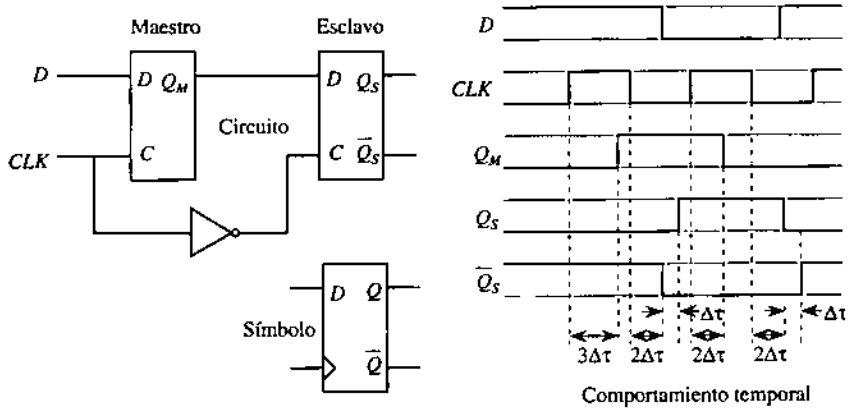


Figura A.48 • Un *flip flop* maestro-esclavo.

Un *flip flop* **activado por nivel** puede cambiar sus estados en forma continua cuando la señal de reloj está en su estado activo (alto o bajo, según como se haya diseñado el *flip flop*). Un *flip flop* **activado por flanco** solo cambia en una transición creciente o decreciente de la señal de reloj. En algunos textos no suele aparecer el símbolo del triángulo en la entrada de reloj para distinguir entre *flip flops* activados por flanco y por nivel, e indican una forma u otra de funcionamiento de alguna manera no muy definida. En la práctica, la notación no es demasiado rigurosa. En este texto se utiliza el símbolo triangular en la entrada de reloj, haciendo ver también el tipo de *flip flop* a partir de la forma en que se lo utiliza.

A.11.4 Flip flops J-K y T

Además de los *flip flops* S-R y D, son muy comunes los *flip flops* **J-K** y **T**. El *flip flop* J-K se comporta en forma similar al *flip flop* S-R, excepto porque cuando las dos entradas valen simultáneamente 1, el circuito conmuta el estado anterior de su salida. El *flip flop* T (por "*toggle*") alterna sus estados, como ocurre en el *flip flop* J-K, cuando sus entradas

están ambas en 1. Los diagramas lógicos y los símbolos de los *flip flops* J-K y T se muestran, respectivamente, en las figuras A.49 y A.50.

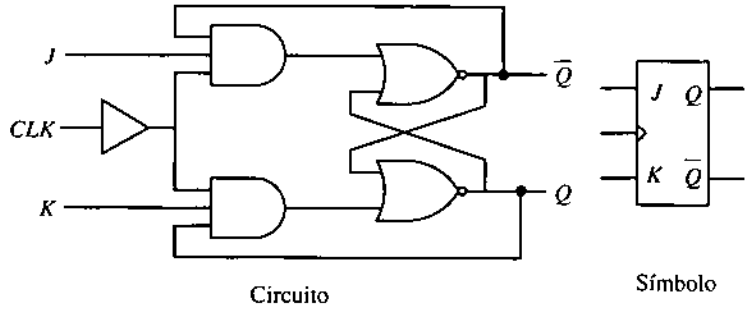


Figura A.49 • Diagrama lógico y símbolo de un *flip flop* J-K básico.

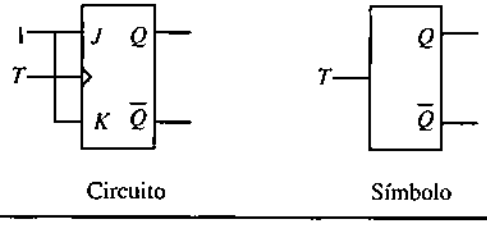


Figura A.50 • Diagrama lógico y símbolo de un *flip flop* T.

Otra vez, puede surgir algún inconveniente cuando en un *flip flop* J-K se tienen las dos entradas *J* y *K* en 1 y se lleva la señal de sincronismo a su estado activo. En esta situación, el *flip flop* puede cambiar de estado más de una vez mientras el reloj está en su estado alto. Esta es otra situación en la que se hace apropiado el uso de un *flip flop* J-K de estructura maestro-esclavo. El esquema de un *flip flop* J-K maestro-esclavo se ilustra en la figura A.51. El problema de la "oscilación infinita" se resuelve con esta configuración, aun cuando la misma crea otro inconveniente. Si se mantiene una entrada en nivel alto durante un tiempo dado mientras la señal de reloj se encuentra activa, aunque fuese porque se encuentre en una transición previa a establecerse, el *flip flop* puede llegar a ver el 1 como si fuera una entrada válida. La situación se resuelve si se eliminan los riesgos en los circuitos que controlan las entradas.

Se puede resolver el problema de la "captura de unos" por medio de la construcción de *flip flops* activados por flanco, en los que el estado de la entrada se analiza solo en las transiciones del reloj (de alto a bajo si el circuito se activa por flanco negativo o de bajo a alto si se trata de un *flip flop* activado por flanco positivo), instantes en los cuales las entradas deberían estar estables.

La figura A.52 ilustra la configuración de un *flip flop* D activado por flanco negativo. Cuando la entrada de reloj está en su estado alto, los circuitos de entrada entregan ceros al *flip flop* S-R principal (de salida). La entrada D puede cambiar una cantidad arbitraria

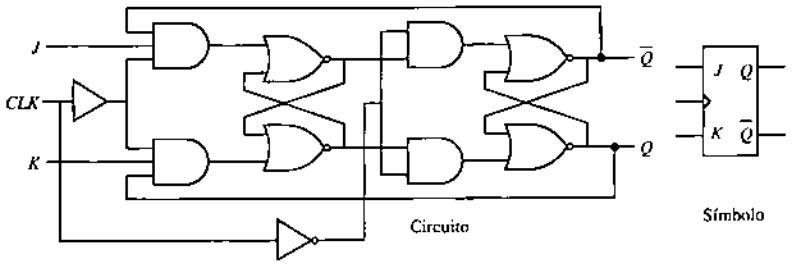


Figura A.51 • Diagrama lógico y símbolo de un *flip flop* J-K maestro-esclavo.

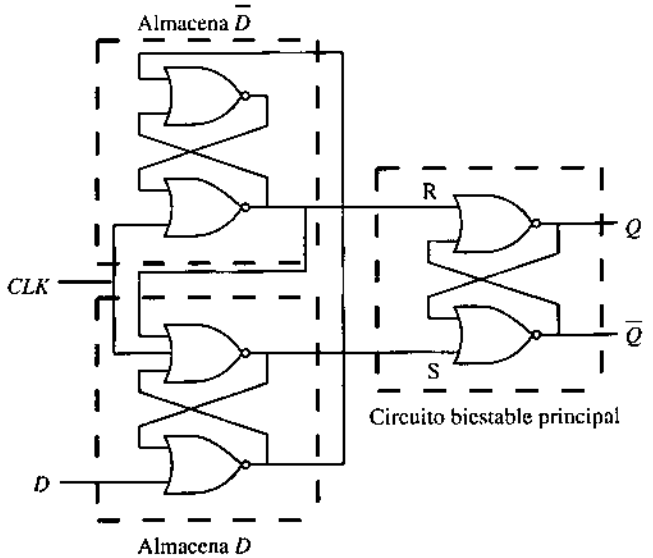


Figura A.52 • *Flip flop* D activado por flanco negativo.

de veces, aún con la señal de sincronismo activa, sin afectar el estado del circuito principal. Cuando el reloj pasa a su estado bajo, el estado del circuito principal solo se ve afectado por los estados estables de los circuitos de entrada. Con el reloj en su estado bajo, aun cuando la entrada D cambie, el circuito principal no se ve afectado.

A.12 Diseño de máquinas de estado

Haciendo nuevamente referencia al modelo clásico de máquina de estados de la figura A.40, se pueden implementar los elementos de retardo por medio de *flip flops* maestro-esclavo, en tanto que la señal de sincronismo puede generarse a través del reloj del sistema. En general, debería haber un *flip flop* en cada lazo de realimentación. Nótese que la identificación de los *flip flops* puede hacerse de cualquier manera en tanto su significado quede claro. En particular, en la figura A.40 las entradas D_i y las salidas Q_i aparecen intercambiadas en relación con las representaciones utilizadas en la sección anterior.

Considérese una máquina de estados finitos de módulo 4, que cuente repetitivamente desde 00 hasta 11. El diagrama en bloques del contador sincrónico se muestra en la figura A.53. La función RESET, de lógica positiva, funciona en forma sincrónica con la señal de reloj. La salida aparece como una secuencia de valores en las líneas q_0 y q_1, a intervalos de tiempo congruentes con la señal de reloj. A medida que se generan las salidas, se genera un nuevo estado s_1s_0 que se realimenta nuevamente hacia la entrada del circuito.

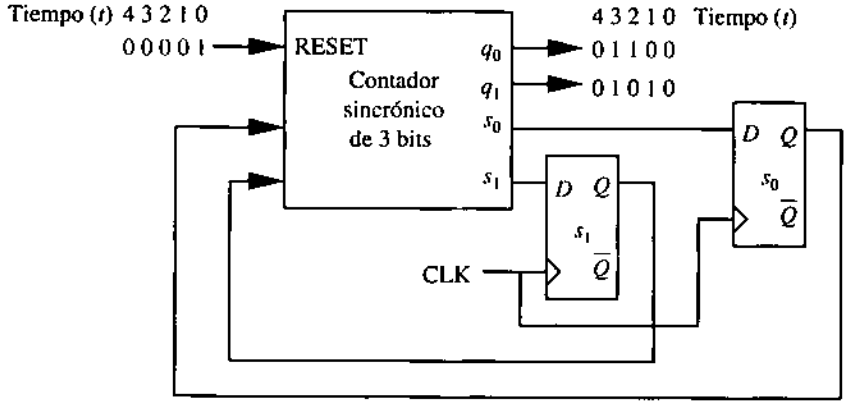


Figura A.53 • Un circuito contador módulo 4.

Se puede diseñar el circuito por medio de la enumeración de todas las posibles condiciones de entrada, creando luego cuatro funciones para las salidas q_1q_0 y los estados s_1s_0. Las funciones correspondientes pueden usarse luego para crear un circuito lógico combinatorio que implemente el contador. Para los dos bits de estado se requieren dos *flip flops*.

¿Cómo se sabe que el lazo de realimentación requiere dos bits de estado? El hecho es que no siempre puede conocerse el número de bits de estado requeridos, por lo que habrá que plantear un enfoque más general para el diseño de una máquina de estados. Para el contador del ejemplo, se puede comenzar definiendo un **diagrama de transiciones entre estados**, tal como el que se muestra en la figura A.54, en el que cada estado representa una de las cuentas entre 00 y 11, en tanto que los arcos representan transiciones entre dichos estados. El estado *A* representa el caso en el que la salida vale 00, y los estados *B*, *C* y *D* representan, respectivamente, los valores 01, 10 y 11 de la cuenta.

Si se supone que la máquina de estados está inicialmente en su estado *A*, existen dos posibles condiciones de entrada: 0 y 1. Si la línea de entrada de RESET está en cero, en el momento de la señal de reloj la máquina avanza al estado *B*, entregando 01 en la salida. Si la línea de entrada está en 1, la máquina de estados permanece en el estado *A* con sus salidas en 00. En forma similar, si se encuentra en el estado *B*, la máquina de estados avanza hacia el estado *C* y entrega 10 si la entrada RESET está en 0, en caso contrario vuelve al estado *A* y entrega en sus salidas el valor 00. Las transiciones desde los estados restantes se interpretan en forma similar.

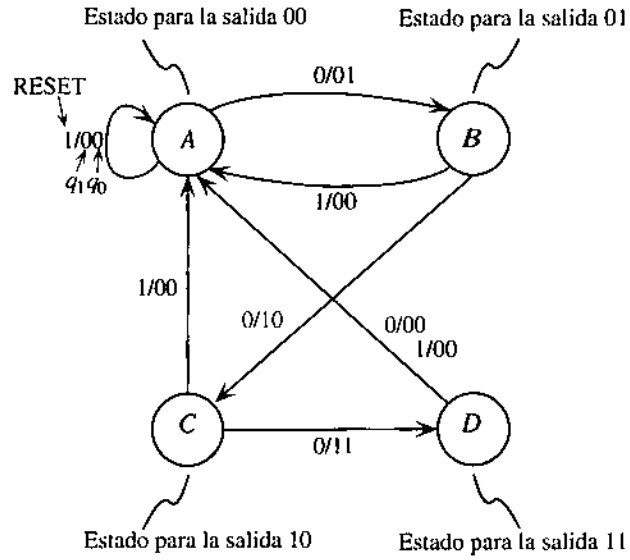


Figura A.54 • Diagrama de transiciones de estados para el contador módulo 4.

Estado actual \ Entrada	*RESET* 0	*RESET* 1
A	*B*/01	*A*/00
B	*C*/10	*A*/00
C	*D*/11	*A*/00
D	*A*/00	*A*/00

Próximo estado Salida

Figura A.55 • Tabla de estados del contador módulo 4.

Una vez creado el diagrama de transiciones entre estados, se lo puede reescribir en forma tabular, dando origen a la **tabla de estados** de la figura A.55. Los estados actuales se muestran a la izquierda de la tabla, en tanto que en la parte superior de la misma se muestran las condiciones de entrada. Las entradas de la tabla corresponden al par de datos próximo estado-valor de salida, los que se obtienen directamente del diagrama de transiciones de la figura A.54. La entrada resaltada corresponde al caso en el que la máquina se encuentra en el estado *B* con su salida en 0. En este caso, el próximo estado es *C* y el valor de la próxima salida es 10.

Luego de creada la tabla de estados, se los codifica en binario. Dado que hay cuatro estados en la tabla, se requiere un mínimo de dos bits para codificar adecuadamente los cuatro estados. Se elige arbitrariamente la codificación $A = 00$, $B = 01$, $C = 10$ y $D = 11$, tras lo cual se reemplaza cada ocurrencia de *A*, *B*, *C* y *D* por sus respectivas codificaciones, como se muestra en la figura A.56. En la práctica, la codificación de estados puede afectar la complejidad de los circuitos resultantes, pero el circuito será correcto desde el punto de vista lógico, cualquiera sea la codificación.

Estado actual (S_t) \ Entrada	*RESET* 0	*RESET* 1
A:00	01/01	00/00
B:01	10/10	00/00
C:10	11/11	00/00
D:11	00/00	00/00

Figura A.56 • Tabla de estado del contador módulo 4 con sus estados asignados.

De la tabla de estados, se pueden obtener tablas de verdad para las funciones de próximo estado y próxima salida, las que se muestran en la figura A.57. Los subíndices en las variables de estado indican relaciones temporales: s_t es el estado actual y s_{t+1} es el estado siguiente. Normalmente, se omiten estos subíndices dado que se entiende que el miembro derecho de la ecuación corresponde a los estados actuales y que los próximos estados aparecen en la izquierda de la misma. Nótese que $s_0(t+1) = q_0(t+1)$ y que $s(t+1) = q_1(t+1)$, por lo que solo deben implementarse $s_0(t+1)$ y $s_1(t+1)$, conectando esas salidas directamente a las salidas $q_0(t+1)$ y $q_1(t+1)$.

RESET $r(t)$	$s_1(t)$	$s_0(t)$	$s_1s_0(t+1)$	$q_1q_0(t+1)$
0	0	0	01	01
0	0	1	10	10
0	1	0	11	11
0	1	1	00	00
1	0	0	00	00
1	0	1	00	00
1	1	0	00	00
1	1	1	00	00

$$s_0(t+1) = \overline{r(t)}\,\overline{s_1(t)}\,\overline{s_0(t)} + \overline{r(t)}s_1(t)\overline{s_0(t)}$$

$$s_1(t+1) = \overline{r(t)}s_1(t)\overline{s_0(t)} + \overline{r(t)}s_1(t)s_0(t)$$

$$q_0(t+1) = \overline{r(t)}\,\overline{s_1(t)}\,\overline{s_0(t)} + \overline{r(t)}s_1(t)\overline{s_0(t)}$$

$$q_1(t+1) = \overline{r(t)}\,\overline{s_1(t)}s_0(t) + \overline{r(t)}s_1(t)\overline{s_0(t)}$$

Figura A.57 • Tabla de verdad y funciones de salida del contador módulo 4.

Por último, se implementan las funciones correspondientes a las salidas y a los próximos estados utilizando compuertas lógicas y *flip flops* D maestro-esclavo, como se ve en la figura A.58.

Ejemplo: un detector de secuencia

Como otro ejemplo, se desea diseñar una máquina que genere un 1 en sus salidas únicamente cuando dos de las tres últimas entradas han sido 1. Por ejemplo, la secuencia de entrada 011011100 genera la secuencia de salida 001111010. Existe una única línea de entrada de datos en serie, pudiendo suponerse que al iniciar el proceso no se ha visto entrada alguna. En este problema se utilizarán *flip flops* D y multiplexores de 8 a 1.

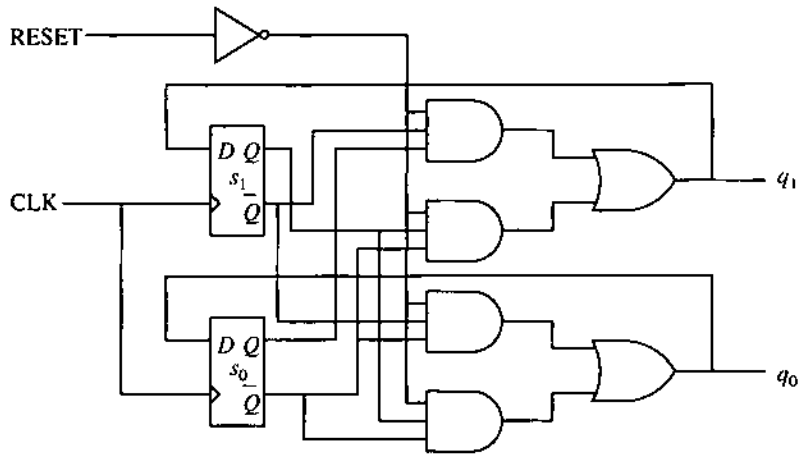


Figura A.58 • Diseño lógico del contador módulo 4.

Se inicia el proceso con la construcción de un diagrama de transiciones entre estados, como se ilustra en la figura A.59. Existen ocho posibles secuencias de tres bits a ser consideradas por el sistema: 000, 001, 010, 011, 100, 101, 110 y 111. El estado *A* es el estado inicial, en el que se supone que la máquina no ha detectado dato alguno de entrada. En los estados *B* y *C* solo se ha visto un valor de entrada, por lo que no puede generarse salida en 1. En los estados *D*, *E*, *F* y *G* se han visto dos entradas, por lo que no puede generarse salida en 1 aun cuando en la entrada al estado *G* ya se han detectado dos unos. La máquina realiza todas las transiciones subsiguientes a través de los estados *D*, *E*, *F* y *G*. El sistema pasa por el estado *D* cuando las dos últimas entradas han sido 00. *E*, *F* y *G* son los estados por los que transcurre el sistema cuando las dos últimas entradas han sido 01, 10 y 11, respectivamente.

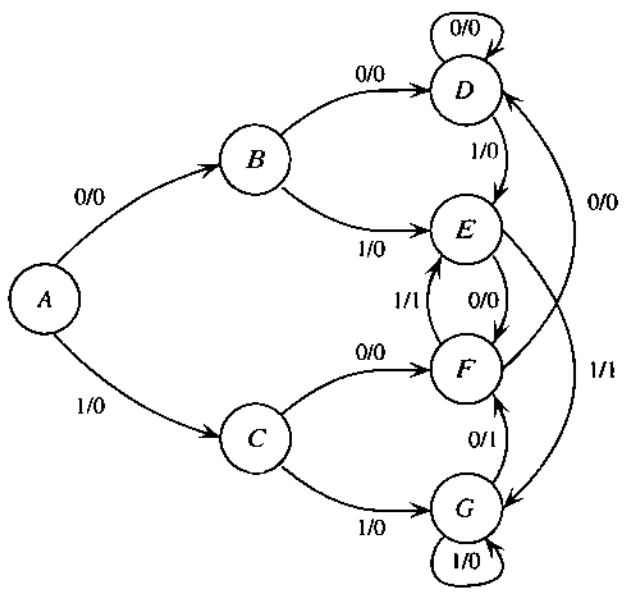


Figura A.59 • Diagrama de transiciones de estados de un detector de secuencia.

Entrada / Estado actual	X = 0	X = 1
A	*B*/0	*C*/0
B	*D*/0	*E*/0
C	*F*/0	*G*/0
D	*D*/0	*E*/0
E	*F*/0	*G*/1
F	*D*/0	*E*/1
G	*F*/1	*G*/0

Figura A.60 • Tabla de estados del detector de secuencia.

El paso siguiente consiste en la creación de la tabla de estados, la que se muestra en la figura A.60, y que se obtiene directamente del diagrama de transiciones. Posteriormente, se realiza la asignación de estados, como la que ilustra la figura A.61. Esta asignación de estados permite crear una tabla de verdad para las funciones que representan las salidas y los estados siguientes, la que se ve en la figura A.61b. Las dos últimas entradas de la tabla corresponden al estado 111, el que, de acuerdo con lo que determina la tabla de la figura A.61a, no puede generarse en la práctica. Por lo tanto, el valor de salida y próximo estado correspondientes no interesan y pueden rotularse como "d", lo que representa un estado irrelevante (*don't care*, en el texto original).

Entrada / Estado actual	X = 0	X = 1
$s_2s_1s_0$	$s_2s_1s_0z$	$s_2s_1s_0z$
A: 000	001/0	010/0
B: 001	011/0	100/0
C: 010	101/0	110/0
D: 011	011/0	100/0
E: 100	101/0	110/1
F: 101	011/0	100/1
G: 110	101/1	110/0

(a)

Entrada y estado en el instante t				Estado futuro y salida en el instante $t+1$			
s_2	s_1	s_0	x	s_2	s_1	s_0	z
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	0	1	1	0
0	0	1	1	1	0	0	0
0	1	0	0	1	0	1	0
0	1	0	1	1	1	0	0
0	1	1	0	0	1	1	0
0	1	1	1	1	0	0	0
1	0	0	0	1	0	1	0
1	0	0	1	1	1	0	1
1	0	1	0	0	1	1	0
1	0	1	1	1	0	0	1
1	1	0	0	1	0	1	1
1	1	0	1	1	1	0	0
1	1	1	0	d	d	d	d
1	1	1	1	d	d	d	d

(b)

Figura A.61 • Asignación de estados y tabla de verdad del detector de secuencia.

Finalmente, se crea el circuito, como el de la figura A.62, en el que se utiliza un total de tres *flip flops* dado que se requiere un *flip flop* por cada variable de estado. Existen tres funciones de estado y una de salida, por lo que se necesitan cuatro multiplexores. Nóte-

se que la selección de s_2, s_1 y s_0 como entradas de control de los multiplexores es arbitraria, en consecuencia, cualquier otro ordenamiento funciona. •

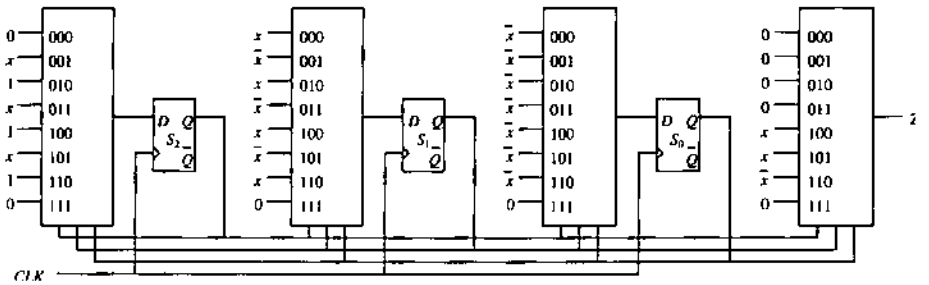


Figura A.62 • Diagrama lógico del detector de secuencia.

Ejemplo: un controlador para una máquina expendedora

En este problema, se diseñará un controlador para una máquina expendedora automática que utilizará *flip flops* D y una PLA, y que se representará como un bloque (como se hizo en la figura A.35). La máquina expendedora acepta tres tipos de monedas: cinco centavos, diez centavos y veinticinco centavos. Cuando la suma de las monedas insertadas es igual o mayor que 20 centavos, la máquina despacha la mercadería, devuelve el exceso de dinero colocado y queda en espera de la próxima operación.

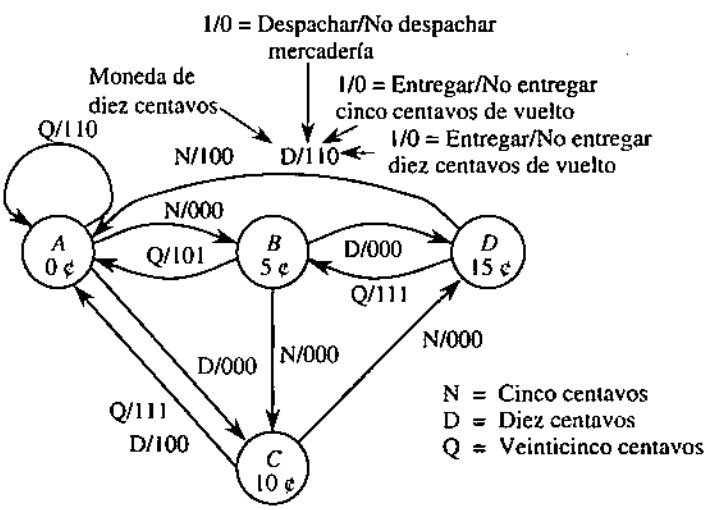


Figura A.63 • Diagrama de transiciones de estados del controlador de máquina expendedora.

Se comienza con la definición del diagrama de transiciones entre estados, el que se muestra en la figura A.63. En el estado *A* no se han colocado monedas, por lo que el dinero

acreditado es 0. Si estando en el estado *A* se colocan 5 o 10 centavos, la máquina de estados produce una transición hacia *B* o hacia *C*, respectivamente. Si lo que se coloca es una moneda de 25 centavos, se procede a despachar la mercadería, a devolver los 5 centavos excedentes y la máquina se mantiene en el estado *A*. Esto se observa en el rótulo *Q*/110 en el lazo cerrado sobre el estado *A*. La expansión de los estados *B* y *C* provoca la aparición del estado *D*, cuya expansión genera el diagrama de estados completo requerido para la máquina expendedora.

Estado actual \ Entrada	N 00	D 01	Q 10
A	*B*/000	*C*/000	*A*/110
B	*C*/000	*D*/000	*A*/101
C	*D*/000	*A*/100	*A*/111
D	*A*/100	*A*/110	*B*/111

(a)

Estado actual \ Entrada	N x_1x_0 00	D x_1x_0 01	Q x_1x_0 10
s_1s_0	s_1s_0 / $z_2z_1z_0$		
A:00	01/000	10/000	00/110
B:01	10/000	11/000	00/101
C:10	11/000	00/100	00/111
D:11	00/100	00/110	01/111

(b)

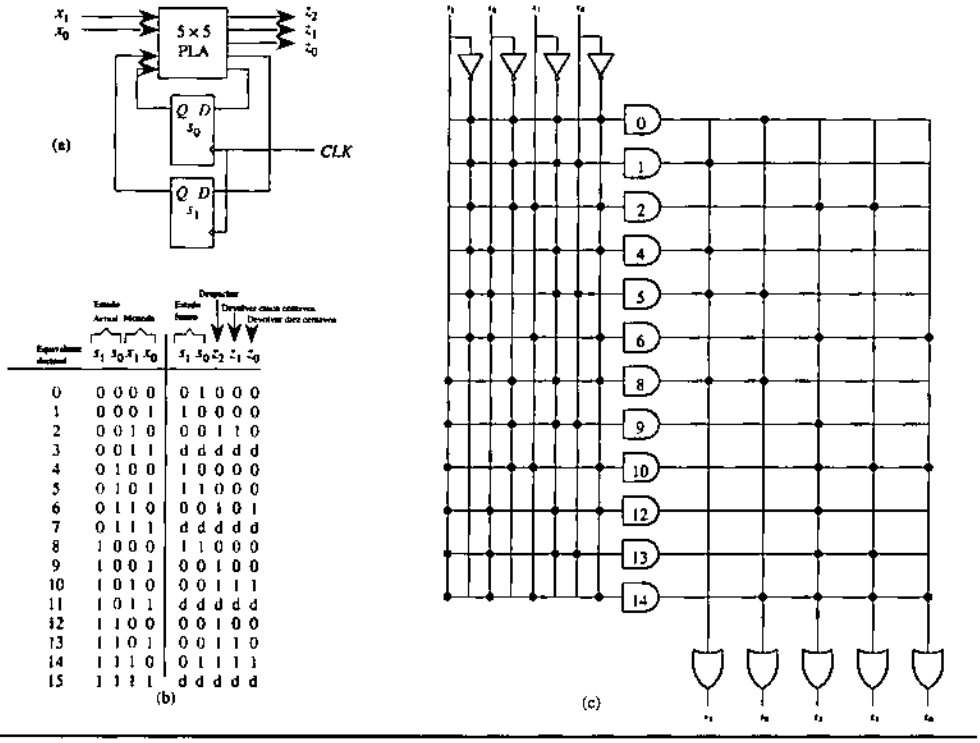
Figura A.64 • (a) Tabla de estados del controlador de la máquina expendedora; (b) asignación de estados del controlador.

Debe observarse la conducta especificada por el diagrama de estados en el caso en que, estando la máquina en el estado *D*, se ingresen 25 centavos. En vez de hacer lo que corresponde, lo que significa despachar la mercadería, devolver 20 centavos y regresar al estado *A*, la máquina entrega la mercadería, devuelve solo 15 centavos y transita hacia el estado *B*. La máquina se queda con un crédito de 5 centavos a la espera del ingreso de otra moneda. En este caso, los autores han aceptado este comportamiento por una cuestión de simplicidad, dado que este comportamiento mantiene bajo el número de estados requerido.

A partir de la máquina de estados finitos se construye la tabla de estados de la figura A.64a. Se hace una asignación arbitraria de los estados y se codifican los símbolos N, D y Q en binario, tal como lo ilustra la figura A.64b. Finalmente, se crea el diagrama circuital, el que se muestra en la figura A.65a. Existen dos bits de estado, por lo que se requieren dos *flip flops* D.

La matriz lógica programable requiere cuatro entradas para los bits de estado actual y los bits representativos de monedas x_1x_0. La matriz genera cinco salidas, para representar los próximos estados y las salidas de entrega de mercadería, de devolución de cinco centavos y de devolución de diez centavos, respectivamente. (Puede suponerse que la entrada de reloj se habilita solo con la ocurrencia de eventos, como el ingreso de una moneda.)

Nótese que el diseño propio de la matriz programable no se ha explicitado en el circuito de la figura A.65a. En estos niveles de complejidad, es habitual el uso de un programa de computadora que genere la tabla de verdad, la que luego se carga en un programa de diseño de la matriz programable. La tabla de verdad y el diseño de la matriz lógica programable, no obstante, podrían generarse a mano, como se observa en las figura A.65b y A.65c.



0	0	0	0	0	0	1	0	0	0
1	0	0	0	1	1	0	0	0	0
2	0	0	1	0	0	0	1	1	0
3	0	0	1	1	d	d	d	d	d
4	0	1	0	0	1	0	0	0	0
5	0	1	0	1	1	1	0	0	0
6	0	1	1	0	0	0	1	0	1
7	0	1	1	1	d	d	d	d	d
8	1	0	0	0	1	1	0	0	0
9	1	0	0	1	0	0	1	0	0
10	1	0	1	0	0	0	1	1	1
11	1	0	1	1	d	d	d	d	d
12	1	1	0	0	0	0	1	0	0
13	1	1	0	1	0	0	1	1	0
14	1	1	1	0	0	1	1	1	1
15	1	1	1	1	d	d	d	d	d

Figura A.65 • (a) Circuito de la máquina de estados finitos; (b) tabla de verdad; (c) implementación del controlador de máquina expendedora que utiliza matrices lógicas programables.

A.13. El modelo Mealy y el modelo Moore

Las salidas de los circuitos de las máquinas de estados finitos analizadas hasta el momento están determinadas por sus estados actuales y sus entradas. Los estados se mantienen por medio de *flip flops* activados por flanco negativo, por ende, solo pueden producirse cambios en los estados en los flancos negativos del reloj. Cualquier cambio que se produzca en las entradas no tiene efecto sobre el estado mientras la entrada de reloj esté baja (inactiva). Las entradas se utilizan directamente en los circuitos de salida sin la participación de *flip flop* alguno. En consecuencia, un cambio de entrada en cualquier instante puede hacer que las salidas cambien, independientemente de si la entrada de reloj está alta o baja. En la figura A.65, el cambio en cualquiera de las entradas x_0 o x_1 puede propagarse hacia las salidas $z_2z_1z_0$, sin importar el nivel de la señal de sincronismo. Esta estructura representa el modelo **Mealy** de una máquina de estados finitos.

En el modelo Mealy, las salidas cambian inmediatamente con los cambios en las entradas, por lo que no hay retardos introducidos por el reloj. En el modelo **Moore** de una máquina de estados finitos, las salidas se encuentran incluidas en los bits de estado, por lo que los cambios en las salidas se producen en los pulsos de reloj posteriores a un cambio en las entradas. Los diseñadores de circuitos usan cualquiera de los dos modelos, cu-

yas descripciones pueden encontrarse fuera de este texto. Esta sección solo pretende señalar las diferencias a través de un ejemplo.

La figura A.66 ilustra un modelo de máquina de estados Moore. La misma cuenta desde 0 hasta 3 en forma repetitiva, tal como lo hace el contador módulo 4 de la figura A.58. La máquina solo cuenta cuando $x = 1$; en caso contrario, la misma mantiene su estado actual. Nótese que las salidas están vinculadas con las variables de estado, de modo tal que no existe camino directo entre una entrada y las salidas sin la participación de un *flip flop*.

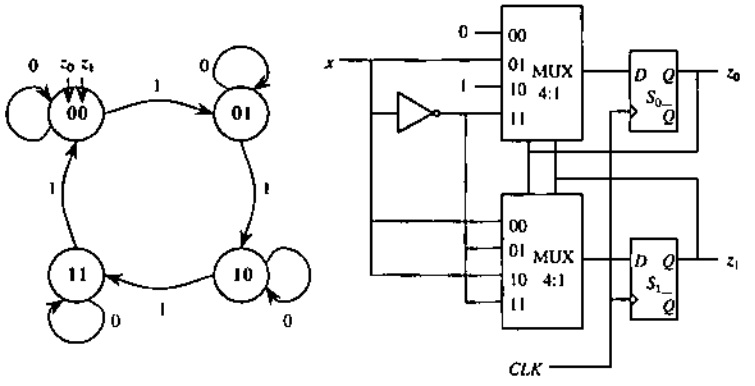


Figura A.66 • Un contador binario según el modelo Moore.

El modelo Mealy puede considerarse más poderoso que el modelo Moore debido a que, en un único ciclo de reloj, un cambio en la salida de una máquina de estados finitos puede llevarse a la entrada de otra máquina de estados finitos, cuya salida cambia y se traslada a la entrada de una tercera máquina, y así continuamente. En cambio, en el modelo Moore, se mantiene una estricta sincronización paso a paso, por lo que no puede producirse esta trasferencia en cascada. Los cambios espurios de las salidas de una máquina de estados finitos tienen, por ende, menos influencia sobre el resto el circuito en el modelo Moore. Esto simplifica el análisis del circuito y la depuración del hardware, por lo que en estas situaciones puede ser preferible el modelo Moore. En la práctica se utilizan ambos modelos. °

A.14 Registros

Un *flip flop* D almacena un único bit de información. Un grupo de N bits, que forman una palabra, puede almacenarse en N *flip flops* D organizados según lo muestra la figura A.67, en la que se ejemplifica con una palabra de 4 bits. Tal disposición de *flip flops* se conoce como "registro". En esta configuración particular, la información de las entradas D_i se carga en el registro cuando las señales de Escritura y Habilitación están en su estado alto, en forma sincronizada con la señal de reloj. Los contenidos del registro pueden leerse en las salidas Q_i solo si la línea de Habilitación está en su estado alto, dado que si esta línea está en su estado bajo, los *buffers* de tres estados ubicados a la salida se encuen-

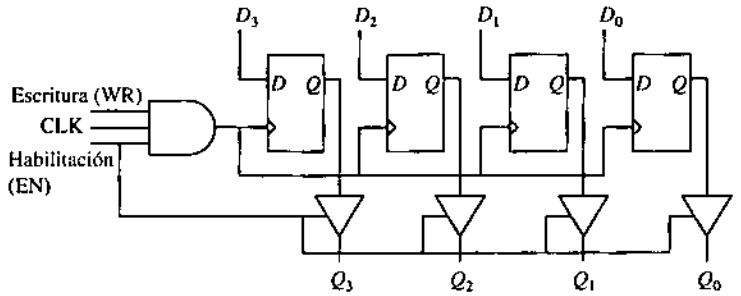


Figura A.67 • Un registro de cuatro bits.

tran eléctricamente desconectados. Se puede simplificar la ilustración, como lo indica la figura A.68, representando nada más que las entradas y salidas.

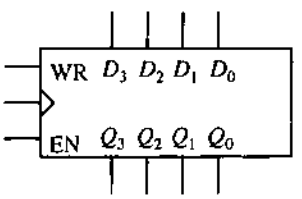


Figura A.68 • Representación abstracta de un registro de cuatro bits.

Un **registro de desplazamiento** copia el contenido de cada uno de sus *flip flops* en el siguiente, mientras recibe nueva información en uno de sus extremos y vuelca el contenido en el otro extremo, lo que se hace posible a través de una configuración en cascada. Considérese el registro de desplazamiento de la figura A.69. El registro puede desplazar información a izquierda o a derecha, aceptar una carga en paralelo o mantenerse sin cambios, todo sincronizado con la señal de reloj. Las prestaciones de carga y lectura en paralelo permiten que el registro de desplazamiento funcione como un **conversor serie-paralelo** o como **conversor paralelo-serie**.

A.15 Contadores

Un **contador** es un tipo distinto de registro, en el que los patrones binarios de salida recorren en secuencia un cierto rango de números binarios. La figura A.70 muestra la configuración de un contador módulo 8 que recorre la secuencia binaria 000, 001, 010, 011, 100, 101, 110 y 111, repitiendo luego la operación. Se utilizan tres *flip flops* J-K en modo T, y cada entrada de reloj se conecta a través de una compuerta Y con la salida Q de la etapa anterior, lo que divide la frecuencia de reloj a la mitad en cada caso. Como resultado se tiene una cadena de *flip flops* T operando a velocidades que difieren en potencias de 2, correspondientes a la secuencia de patrones binarios que va de 000 a 111.

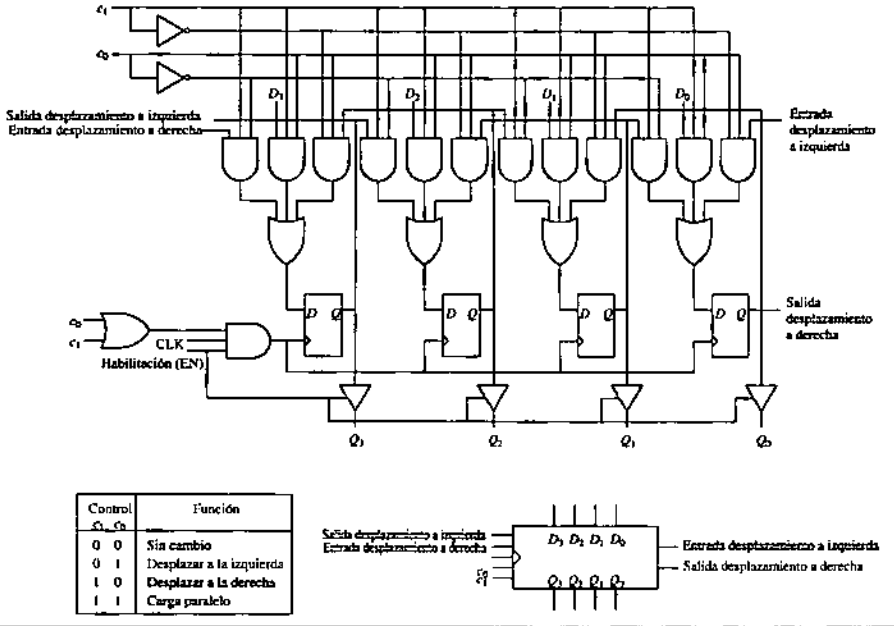


Figura A.69 • Distribución interna y diagrama en bloques de un registro de desplazamiento bidireccional con capacidad de carga y lectura en paralelo.

Nótese el agregado de una línea asincrónica de RESET, de funcionamiento activo en estado bajo, que lleva el contador a 000 en forma independiente de los estados del reloj o de la línea de Habilitación. Excepción hecha del *flip flop* menos significativo, los restantes cambian sus estados de acuerdo con los cambios de estado que se producen en sus vecinos a derecha, en lugar de estar sincronizados con la señal de reloj. El funcionamiento es similar al del contador módulo 4 de la figura A.58, pero el diseño se puede extender más fácilmente a tamaños mayores debido a que no se lo trata como una máquina de estados con propósitos de diseño, en las que se enumeran todos los estados. Así y todo, es una máquina de estados finitos.

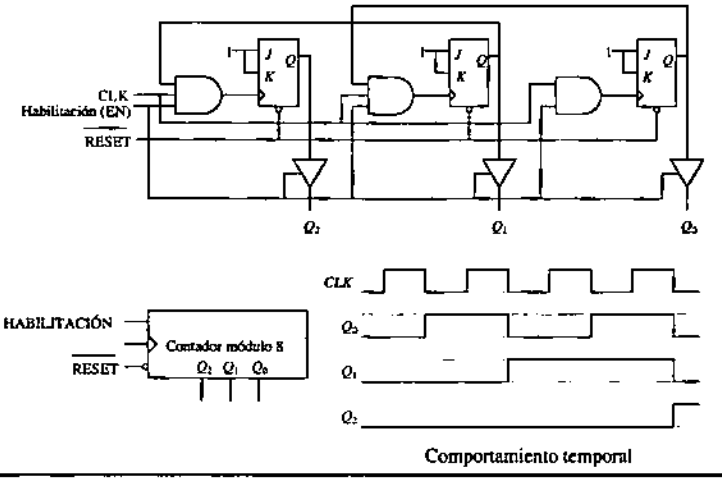


Figura A.70 • Un contador de módulo 8.

Resumen

En teoría, cualquier función booleana puede representarse como una tabla de verdad, la cual puede ser transformada en una ecuación booleana de dos niveles e implementarse con compuertas lógicas. En la práctica, los conjuntos de compuertas lógicas pueden agruparse para formar componentes integrados en mediana escala, los que contienen algunas pocas unidades o decenas de compuertas lógicas. Los circuitos multiplexores y las matrices lógicas programables son dos tipos de componentes de integración en mediana escala que pueden usarse para la implementación de funciones. Los circuitos decodificadores se usan para habilitar una única línea de salida en función de la combinación binaria presente en las entradas del mismo, lo que traduce una codificación lógica en una posición espacial. Asimismo, existen otros tipos de circuitos MSI. En la utilización de circuitos integrados en mediana escala se puede abstraer la complejidad del circuito al nivel de las compuertas requeridas. Los circuitos integrados en alta y muy alta escala de integración (LSI y VLSI) también pueden abstraer la complejidad de circuitos de mayores niveles.

Una máquina de estados finitos difiere de un circuito combinatorio en el hecho de que las salidas del circuito combinatorio en cualquier momento solo son función de las entradas en ese momento, en tanto que las salidas de la máquina de estados finitos también son función de la historia previa de sus entradas.

Para lectura posterior

Las contribuciones de C. E. Shannon al álgebra de conmutación se basan en los trabajos de G. Boole, y constituyen la base de la teoría de conmutación tal como se la conoce actualmente. La cantidad de aportes al álgebra de Boole es tan vasta que se hace complicado detallarla aquí. Z. Kohavi es una buena referencia general para circuitos combinatorios y secuenciales. El aporte hecho por E. S. Davidson desarrolla un método para la descomposición de circuitos basados en compuertas NAND, lo que resulta interesante debido a que algunas computadoras están íntegramente diseñadas sobre la base de compuertas NAND.

La literatura de Xilinx cubre la filosofía y los aspectos prácticos de las matrices de compuertas y describe configuraciones de su línea de **matrices de compuertas programables por el usuario** (FPGA, *field programmable gate arrays*).

Algunos textos señalan la diferencia entre un *flip flop* y un *latch*. A. Tanenbaum hace una distinción entre ambos planteando que un circuito *flip flop* es aquel que se activa por flancos, en tanto que un *latch* se activa por niveles. Esta definición puede ser correcta, pero, en la práctica, las dos palabras suelen intercambiarse a menudo, por lo que cualquier distinción entre ambos términos se enturbia.

Boole, G., *An Investigation of the Laws of Thought*, Dover Publications, Inc., 1854.

Davidson, E. S., "An algorithm for NAND decomposition under network constraints", en: *IEEE Transactions on Computers*, C-18, 1098, 1979.

Kohavi, Z., *Switching and Finite Automata Theory*, 2ª ed., McGraw Hill, 1978.

Shannon, C. E., "A symbolic analysis of relay and switching circuits", en: *Trans. American Institute of Electrical Engineers*, vol. 57, 1938, p.p. 713-723.

Shannon, C. E., "The sintesis of two terminal switching circuits", en: *Bell System Technical Journal*, vol. 28, 1949, p.p. 59-98.

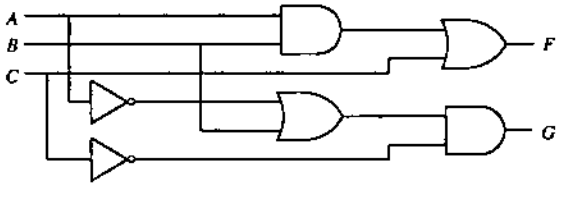
Tanenbaum, A., *Structured Computer Organization*, 4ª ed., Prentice Hall, 1999. (Traducción al español disponible: *Organización de computadoras, un enfoque estructurado*, 4ª ed., Prentice Hall, 2000.)

Xilinx, *The Programmable Gate Array Data Book*, Xilinx, Inc., 2100 Logic Drive, San José, California, 1992.

Problemas

A.1 La figura A.13 muestra la implementación de una compuerta O a través de una compuerta NAND e inversores. La figura A.14 muestra inversores implementados con compuertas NAND. Representar el diagrama lógico de una compuerta Y realizada íntegramente con compuertas NAND.

A.2 Dibujar diagramas lógicos de cada elemento del conjunto de compuertas [Y, O, NO] utilizando únicamente compuertas NOR.



A.3 Dado el circuito lógico indicado, construir la tabla de verdad que describe su comportamiento.

A.4 Construir la tabla de verdad para una función XOR de tres variables.

A.5 Indicar la cantidad de entradas de compuertas del codificador de prioridad de la figura A.32. Incluir los inversores en el cálculo.

A.6 Diseñar un circuito que implemente la función *f* indicada a continuación utilizando compuertas Y, O, NO:

$$f(A, B, C) = \bar{A}BC + \bar{A}\bar{B}\bar{C} + AB\bar{C}$$

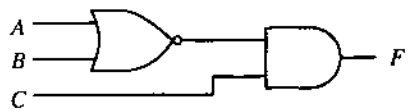
A.7 Diseñar un circuito que implemente la función *g* indicada a continuación utilizando compuertas Y, O, NO. No debe intentarse modificar la forma en que se expresa la función.

$g(A, B, C, D, E) = A(BC + \bar{B}\bar{C}) + B(CD + E)$

A.8 ¿Son equivalentes las funciones *f* y *g* siguientes? Justificar la respuesta.

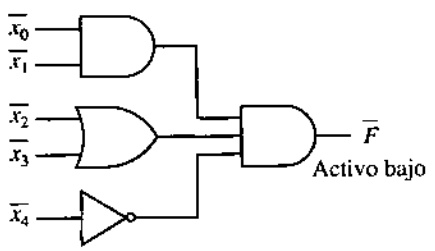
$f(A, B, C,) = ABC + \bar{A}B\bar{C}$ $\qquad$ $g(A, B, C,) = (A \oplus C)B$

A.9 Describir el comportamiento de la función *F* del siguiente circuito por medio de una ecuación lógica. Presentarla en forma suma de productos (sin paréntesis).



A.10 Un **comparador** de cuatro bits es un circuito lógico que recibe como entradas dos palabras de cuatro bits cada una, entregando una única salida. La salida es 0 si las palabras son iguales y 1 si son distintas. Diseñar un comparador para dos números de cuatro bits con cualquiera de las compuertas vistas en este apéndice. *Sugerencia*: Considerar el comparador de dos números de cuatro bits a partir de alguna combinación de cuatro comparadores de un bit.

A.11 Redibujar el circuito siguiente de modo que se obtenga una adecuada coincidencia de inversores. Las barras que van encima de los nombres de variables y función indican lógica negativa.



A.12 Utilizar dos multiplexores de cuatro entradas de datos para implementar las siguientes funciones.

A	B	F_0	F_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

A.13 Implementar la función mayoría con un multiplexor de cuatro entradas de datos.

A.14 Utilizar un decodificador de 2 a 4 y una compuerta O para implementar la XOR de dos variables *A* y *B*.

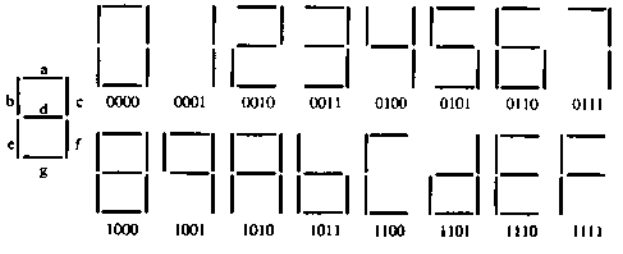
A.15 Dibujar un diagrama lógico que, a través de un decodificador y dos compuertas O, implemente las funciones *F* y *G* siguientes. No omitir la identificación de todas las líneas del diagrama.

$$F(A, B, C) = \bar{A}B\bar{C} + \bar{A}\bar{B}C + A\bar{B}C + \bar{A}BC$$
$$G(A, B, C) = \bar{A}B\bar{C} + ABC$$

A.16 Utilizando únicamente multiplexores de 2 a 1, diseñar un circuito que implemente la función de un multiplexor de 8 a 1. Representar el diseño por medio de un diagrama lógico, sin omitir la identificación de todas las líneas.

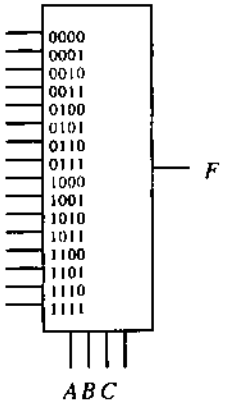
A.17 Dado que cualquier circuito combinatorio puede construirse usando solo compuertas NAND de dos entradas, las mismas suelen recibir el nombre de compuertas lógicas universales. La compuerta NOR de dos entradas también es una compuerta lógica universal, no así las compuertas Y y O. Dado que una compuerta NAND de dos entradas se puede construir usando multiplexores de cuatro entradas de datos (en realidad, con un multiplexor de cuatro entradas), cualquier circuito combinatorio puede construirse usando solo multiplexores de cuatro entradas. En consecuencia, el multiplexor de cuatro entradas de datos es también un dispositivo universal. Demostrar que un demultiplexor de una entrada a dos salidas es un dispositivo universal, por medio de la construcción de una compuerta NAND de dos entradas, usando solo demultiplexores de dos salidas. Dibujar el diagrama lógico. *Sugerencia*: Obtener la compuerta NAND a partir de una compuerta Y y un inversor, implementados cada uno de ellos con demultiplexores de dos salidas.

A.18 Se muestra un indicador de siete segmentos, como el que se puede encontrar en una calculadora. Los siete segmentos se identifican como *a-g*. Diseñar un circuito que tome como entrada un número binario de cuatro bits y que entregue como salida la señal de control del segmento *b*. Un 0 en la salida lo apaga, en tanto que un 1 lo enciende. Obtener la tabla de verdad y una implementación que utilice un único multiplexor, sin lógica adicional. Identificar todas las líneas del multiplexor.



A.19 Implementar la función *F* correspondiente a la siguiente tabla de verdad utilizando el multiplexor de 16 entradas que se indica. Identificar todas las líneas, incluyendo la línea de control no utilizada.

A	*B*	*C*	*F*
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1



A.20 **Un codificador** toma 2^N entradas binarias, de las que, en cualquier momento, solo una es 1 en tanto que las demás son 0, y produce una salida de *N* bits, codificada en binario, que indica cuál de las entradas está en estado alto. Generar una tabla de verdad para un codificador de cuatro entradas y dos salidas, en el cual las cuatro entradas *A*, *B*, *C* y *D* se codifican en dos salidas *X* e *Y*. *A* y *X* son los bits más significativos.

A.21 Considerar un circuito lógico combinatorio con tres entradas *a*, *b* y *c*, y seis salidas *u*, *v*, *w*, *x*, *y*, *z*. La entrada es un número sin signo entre 0 y 7 y la salida es el cuadrado de la entrada. El bit más significativo de la entrada es *a*, y el de la salida es *u*. Crear la tabla de verdad de las seis funciones.

A.22 Considerar la función *f(a,b,c,d)* que adopta el valor 1 solo si el número de unos en *b* y *c* es mayor o igual que la cantidad de unos en *a* y *d*.

a. Escribir la tabla de verdad de la función *f*.

b. Implementar la función *f* con un multiplexor de ocho entradas.

A.23 Diseñar la tabla de verdad para un comparador ternario (base 3) de un dígito. Las entradas ternarias son *A* y *B*, cada una de las cuales es de un dígito. La salida *Z* es 0 para $A < B$, 1 para $A = B$ y 2 para $A > B$. Utilizando esta tabla de verdad como guía, reescribir la tabla en binario, asignando $(0)_3 = (00)_2$, $(1)_3 = (01)_2$ y $(2)_3 = (10)_2$.

A.24 Demostrar el teorema del consenso para tres variables utilizando inducción completa.

A.25 Utilizar las propiedades del álgebra de Boole para demostrar en forma algebraica el teorema de DeMorgan.

A.26 ¿Puede fabricarse un *flip flop* S-R usando dos compuertas XOR realimentadas? Justificar la respuesta.

A.27 Modificar el diagrama de transiciones de estados de la máquina expendedora para que su comportamiento sea más apropiado (devolver todo el dinero excedente) cuando, encontrándose en el estado D, se ingresa una moneda de $ 0,25.

A.28 Crear el diagrama de estados de una máquina de estados finitos que ordene dos palabras binarias *A* y *B*, las que llegan con su bit más significativo primero, en dos salidas *GE* y *LT*. Si *A* es mayor o igual que *B*, *A* aparece en la salida *GE* y *B* en la salida *LT*. Si *B* es mayor que *A*, entonces *B* aparece en *GE* y *A* aparece en *LT*.

A.29 Diseñar un circuito que produzca un 1 en la salida *Z* cuando la entrada *X* cambie de 0 a 1 o de 1 a 0, y que se mantenga en cero en toda otra ocasión. Para el estado inicial deberá suponerse que la última entrada vista fue un cero. Por ejemplo, si la secuencia de entrada es 00110 (de izquierda a derecha), la secuencia de salida es 00101. Representar el diagrama de estados, la tabla de estados, la asignación de estados y el circuito final que deberá implementarse utilizando multiplexores.

A.30 Diseñar una máquina de estados finitos cuya salida es 1 cada vez que los últimos tres valores de entrada sean 011 o 110. Indicar solo la tabla de estados. No representar el circuito.

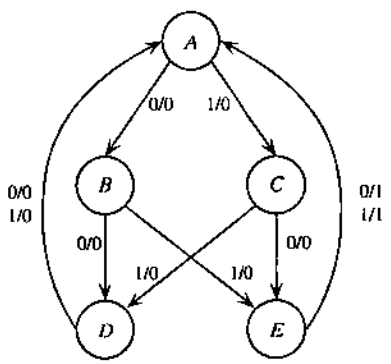
A.31 Diseñar una máquina de estados finitos que tome dos palabras binarias *X* e *Y* en formato serie, bit menos significativo primero, y entregue una salida *Z* de un bit, que es 1 cuando *X* es mayor que *Y*, y es cero cuando *X* es menor o igual que *Y*. Para el estado inicial, suponer que *X* = *Y*. Por lo tanto, *Z* vale 0 hasta que *X* > *Y*. El circuito indicado muestra un ejemplo de secuencias de entrada y salida.

A.32 Crear un diagrama de transiciones de estados para una máquina de estados finitos que ordena dos entradas ternarias, dígito más significativo primero, hacia dos salidas ternarias *GE* y *LT*. Si *A* es mayor o igual a *B*, *A* aparece en la salida *GE* y *B* aparece en la salida *LT*. En caso contrario, *B* aparece en la salida *GE* y *A* aparece en la línea *LT*. Se muestra una secuencia de entrada y salida como ejemplo. Utilizar los símbolos ternarios 0, 1 y 2 al rotular los arcos del diagrama.

Entrada A:	0 2 1 1 2 0 1 2
Entrada B:	0 2 1 2 0 2 1 1
Salida GE:	0 2 1 2 0 2 1 1
Salida LT:	0 2 1 1 2 0 1 2
Tiempo:	0 1 2 3 4 5 6 7

A.33 Crear un diagrama de estados de una máquina que calcula el bit z de paridad par para una entrada de dos bits $x_1\ x_0$. La máquina entrega un 0 en su salida cuando todas las entradas de dos bits recibidas previamente tienen un número par de unos, y entrega un 1 en caso contrario. Para el estado inicial, suponer que la máquina comienza con paridad par.

A.34 Diseñar un circuito para la máquina de estados descripta por el siguiente diagrama de estados, usando *flip flops* D, un único decodificador y compuertas O. Para la asignación de estados, utilizar el número binario correspondiente a la posición de cada letra en el alfabeto, empezando desde 0. Por ejemplo, *A* está en la posición 0, por lo que su asignación es 000, *B* corresponde a la posición 1, por lo que se le asigna 001, etcétera.



A.35 Redibujar el circuito de la figura A.16 utilizando compuertas O e Y de dos entradas.

A.36 Suponer que se requiere implementar una compuerta Y de N entradas, usando solo compuertas Y de tres entradas. ¿Cuál es el mínimo número de retardos de compuertas requerido para implementar la compuerta Y de N entradas? Una compuerta Y tiene un retardo de 1, dos compuertas en cascada tienen, en conjunto, un retardo de 2, etcétera.

Apéndice B

Simplificación de circuitos lógicos

B.1 Reducción de lógica combinatoria y de lógica secuencial

El apéndice A planteó su enfoque básicamente en los aspectos funcionales de los circuitos lógicos digitales. Casi no se tuvo en cuenta la posibilidad de que hubiese más de una forma de diseñar un circuito, siendo algunos diseños mejores que otros en términos de la cantidad de componentes (esto es, cantidad y tamaño de las compuertas lógicas).

En este apéndice se enfoca un método sistemático que permita reducir la cantidad de componentes de un diseño. El primer punto será el análisis de cómo reducir el tamaño de las expresiones lógicas combinatorias, de quienes dependen la cantidad y tipo de compuertas lógicas con las que se implementa un circuito digital. Luego, se analiza la forma de reducir la cantidad de estados en máquinas de estados finitos, explorando algunas áreas de diseño de las máquinas de estados finitos que impactan sobre la cantidad y el tamaño de las compuertas lógicas utilizadas en su implementación.

B.2 Reducción de las expresiones de dos niveles

En muchos casos, las expresiones canónicas **suma de productos** y **producto de sumas** de una función no son mínimas en términos de su tamaño y cantidad de elementos. Dado que una ecuación booleana más pequeña se traduce en un circuito con menor cantidad de compuertas, la reducción de la ecuación es un tema importante cuando se debe tener en cuenta la complejidad circuital.

En las secciones siguientes se describen tres métodos para la reducción de ecuaciones booleanas: la **simplificación algebraica**, el **método de simplificación por medio de los mapas de Karnaugh (mapas K)** y el **método de simplificación tabular**. El método algebraico es la base de los otros dos métodos. Es también el método más abstracto, dado que se apoya solo en los teoremas del álgebra de Boole.

El método de los mapas K y el método tabular son, de hecho, implementaciones de papel y lápiz del método algebraico. Se los analiza para que el lector pueda visualizar el procedimiento de reducción y, por ende, tener una idea más clara acerca de su funciona-

miento. Estos procedimientos se pueden usar en forma efectiva para minimizar funciones del orden de seis o menos variables. Para funciones más grandes, generalmente resulta más efectiva la solución ofrecida por los programas de diseño asistido por computadora (CAD, *Computer Aided Design*).

B.2.1 El método algebraico

El método algebraico utiliza las propiedades del álgebra de Boole, planteadas en la sección A.5, en una forma sistemática que permite reducir el tamaño de las expresiones. Considérese la ecuación booleana de la función mayoría, que se reproduce del apéndice A.

$$F = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC \tag{B.1}$$

Para reducir la ecuación a una forma más simple, pueden aplicarse las propiedades del álgebra de Boole, como se muestra en las ecuaciones B.2 a B.4:

$$F = \bar{A}BC + A\bar{B}C + AB\,(\bar{C} + C) \qquad \text{Propiedad distributiva} \tag{B.2}$$

$$F = \bar{A}BC + A\bar{B}C + AB\,(1) \qquad \text{Propiedad del complemento} \tag{B.3}$$

$$F = \bar{A}BC + A\bar{B}C + AB \qquad \text{Propiedad de identidad} \tag{B.4}$$

El circuito correspondiente a la ecuación B.4 se muestra en la figura B.1. Si se lo compara con el circuito utilizado en la figura A.16, la cantidad de compuertas se reduce de 8 a 6 y la cantidad de entradas se reduce de 19 a 13.

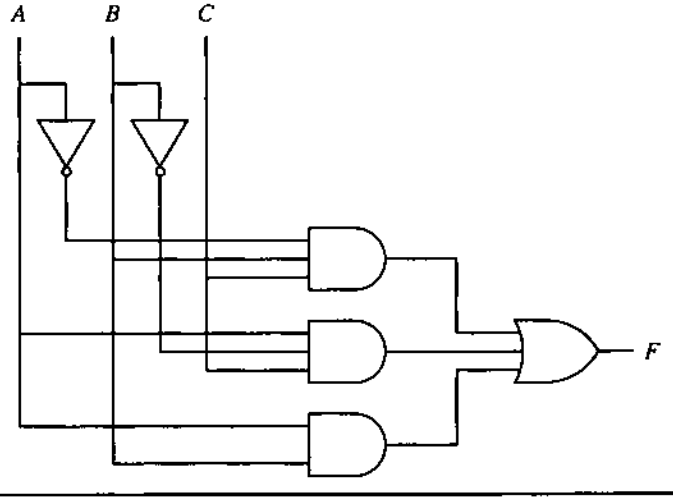


Figura B.1 • Circuito reducido para la función mayoría.

La función puede reducirse aun más a partir de la ecuación B.4. Si se aplica la propiedad de idempotencia, se obtiene la ecuación B.5, en la que reaparece el término mínimo *ABC*.

$$F = \bar{A}BC + A\bar{B}C + AB + ABC \qquad \text{Propiedad de idempotencia} \qquad (B.5)$$

Si se vuelven a aplicar las propiedades distributiva, de complemento y de identidad, se obtiene una ecuación más simple, según puede verse:

$$F = \bar{A}BC + AC(\bar{B} + B) + AB \qquad \text{Propiedad distributiva} \qquad (B.6)$$

$$F = \bar{A}BC + AC(1) + AB \qquad \text{Propiedad del complemento} \qquad (B.7)$$

$$F = \bar{A}BC + AC + AB \qquad \text{Propiedad de identidad} \qquad (B.8)$$

La ecuación B.8 tiene una cantidad de 11 entradas, aún menor que las anteriores implementaciones. Una nueva iteración del método permite reducir nuevamente la función, según se puede ver a continuación:

$$F = \bar{A}BC + AC + AB + ABC \qquad \text{Propiedad de idempotencia} \qquad (B.9)$$

$$F = BC(\bar{A} + A) + AC + AB \qquad \text{Propiedad distributiva} \qquad (B.10)$$

$$F = BC(1) + AC + AB \qquad \text{Propiedad del complemento} \qquad (B.11)$$

$$F = BC + AC + AB \qquad \text{Propiedad de identidad} \qquad (B12)$$

La ecuación B.12 está expresada en su forma mínima de dos niveles y no puede seguir siendo simplificada.

B.2.2 El método de los mapas K

El método de los mapas K es, en concreto, una técnica gráfica que puede usarse para visualizar los términos mínimos de una función junto con las variables que les son comunes. Las variables comunes a más de un término mínimo son candidatas a su eliminación, según ya se ha analizado anteriormente. La base del mapa K es el diagrama de Venn, originalmente diseñado para la visualización de conceptos en la teoría de conjuntos.

El diagrama de Venn para variables binarias es un rectángulo que representa el universo binario en formato suma de productos. La figura B.2 ilustra un diagrama de Venn para tres variables *A*, *B* y *C*. Dentro del universo, cada variable se representa con un círculo. Dentro de su círculo, la variable tiene el valor 1 y fuera de él adopta el valor 0. Las intersecciones representan los términos mínimos, como se ve en la figura.

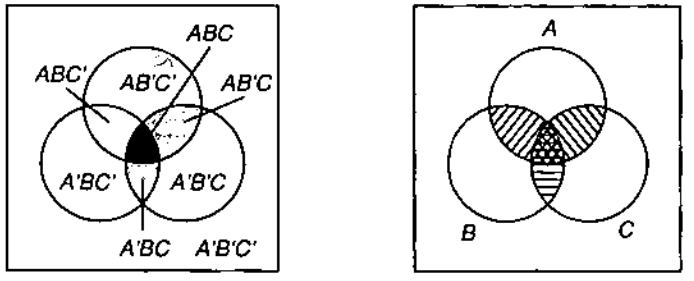


Figura B.2 • Representación de tres variables binarias (izquierda) y de la función mayoría (derecha) a través de los diagramas de Venn.

Las regiones sombreadas adyacentes son candidatas a ser reducidas, dado que difieren exactamente en una variable. En la figura, la región *ABC* puede combinarse con cada una de las tres regiones adyacentes para obtener una forma reducida de la función mayoría. El mapa K es solo una transformación topológica del diagrama de Venn en la que se preserva la relación entre las variables. En el mapa K, al igual que en los diagramas de Venn, los términos mínimos que difieren en una única variable se ubican adyacentes unos a otros.

La figura B.3 muestra el mapa K para la función mayoría. Cada celda del mapa corresponde a una entrada de la tabla de verdad de la función, y dado que hay ocho entradas en la tabla de verdad, hay ocho celdas en el mapa K correspondiente. Se coloca un 1 en cada celda correspondiente a una combinación de variables que hace cierta la función. En las celdas restantes se coloca un 0, el que, por claridad, puede no representarse en el mapa, tal como se ilustra. La definición de filas y columnas se establece según un **código Gray**, en el que se logra un único cambio de variables entre celdas adyacentes, en cada una de las dimensiones.

C \ AB	00	01	11	10
0			1	
1		1	1	1

Figura B.3 • Mapa K para la función mayoría.

Los unos adyacentes del mapa K satisfacen las condiciones requeridas para aplicar la propiedad de complemento del álgebra de Boole. Dado que en el mapa K de la figura B.3 existen unos adyacentes, puede lograrse una simplificación. Las agrupaciones de celdas adyacentes se realizan en cantidades de términos mínimos que deben ser potencias de dos, tales como 1, 2, 4 y 8. Estos grupos se conocen con el nombre de **implicantes primos**.* Las va-

* *N. de T.*: Se denomina **implicantes primos** a aquellas agrupaciones de términos mínimos, en grupos de 1, 2, 4, 8, etc., que no pueden incluirse en agrupaciones de mayor tamaño. Para aquellas agrupaciones de términos adyacentes que no sean implicantes primos, se reserva en general el nombre de **implicante o cubo**.

riables booleanas se van eliminando a medida que se logra el aumento de tamaño de estos grupos, a partir del grupo formado por un solo término, por lo que se utilizan los grupos de mayor tamaño que puedan obtenerse. Con el objeto de mantener la propiedad de adyacencia, la forma del grupo debe ser siempre rectangular, y cada grupo debe contener un número de celdas que corresponda a una potencia entera de dos.

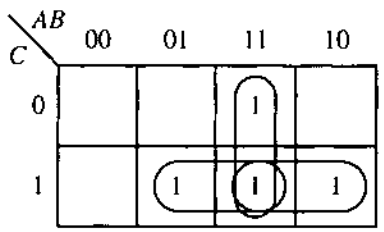


Figura B.4 • Agrupamientos adyacentes para la función mayoría.

El proceso de reducción se inicia creando grupos de unos que pueden estar contenidos en un grupo más grande, avanzando hacia grupos más grandes, hasta que todas las celdas que contengan un uno hayan sido cubiertas al menos una vez. El criterio de adyacencia es crucial, dado que la búsqueda consiste en encontrar grupos de términos que difieran de forma tal que puedan reducirse aplicando las propiedades booleanas de identidad y complemento, como se ve en la ecuación B.13.

$$ABC + AB\bar{C} = AB(C + \bar{C}) = AB(1) = AB \tag{B.13}$$

Para el caso de la función mayoría, es posible formar tres grupos de dos términos, como se ve en la figura B.4. Cada celda con un uno tiene al menos una celda vecina con un uno, por lo que no quedan grupos de un solo término mínimo. Al analizar los grupos formados por dos elementos, se observa que todos los elementos unitarios se encuentran cubiertos por grupos de dos elementos. Una de las celdas se incluye en las tres agrupaciones, lo que se admite, en el proceso de reducción, basado en el principio de idempotencia. El principio de complemento elimina la variable que difiere entre celdas, obteniéndose la ecuación mínima resultante, indicada en la ecuación B.14.

$$M = BC + AC + AB \tag{B.14}$$

El término BC se obtiene a partir del par $(ABC + \bar{A}BC)$, el que se reduce a $BC(\bar{A} + A)$, y luego a BC. El término AC se obtiene en forma similar a partir del grupo $(ABC + A\bar{B}C)$, así como el término AB se obtiene a partir de $(ABC + AB\bar{C})$. El circuito correspondiente se muestra en la figura B.5. La cantidad de compuertas, con relación al circuito de la figura A.16, se reduce de 8 a 4, en tanto que la cantidad de entradas se reduce de 19 a 9.

Para analizar más de cerca el método de simplificación, considérese qué ocurriría si se inicia la simplificación a partir de los grupos más grandes. La figura B.6 muestra la

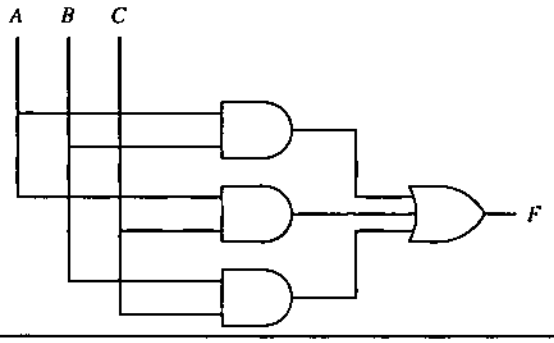


Figura B.5 • Circuito mínimo Y-O de la función mayoría.

aplicación de ambos caminos a un mismo mapa K. La reducción de la izquierda se obtiene empezando a trabajar con los unos que no pueden ser incluidos en grupos mayores, método que ha sido usado con anterioridad. El agrupamiento se realiza en el orden indicado por los números. Se obtiene un total de cuatro grupos, cada uno con dos términos mínimos. La simplificación de la derecha se obtiene iniciando el procedimiento por los grupos más grandes. Se obtienen aquí cinco grupos, uno que incluye a cuatro términos adyacentes y cuatro de dos términos. Así, es posible ver que si se inicia el proceso desde los grupos más grandes puede no obtenerse la ecuación mínima. Ambas ecuaciones de la figura B.6 describen la misma función, y ambos circuitos obtenidos desde ellas serán lógicamente correctos, pero solo una de ellas será mínima.

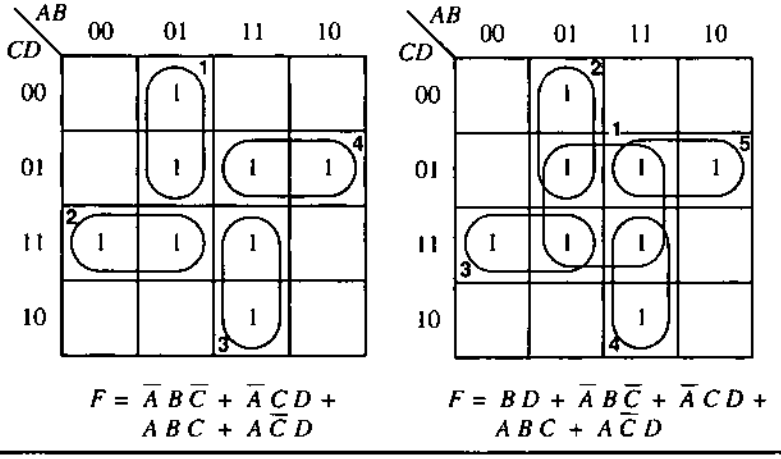


Figura B.6 • Dos agrupaciones de un mismo mapa: la de la izquierda es mínima, la de la derecha no lo es.

Como otro ejemplo, considérese el mapa K de la figura B.7. Los extremos del mapa K se envuelven tanto en forma horizontal como vertical, siendo las cuatro esquinas lógicamente adyacentes. La ecuación mínima correspondiente se indica en la misma figura.

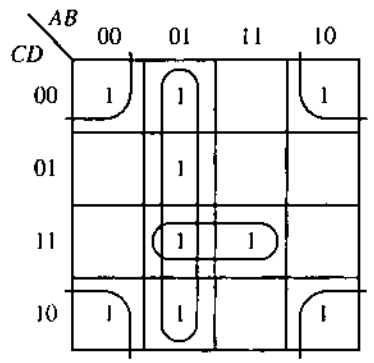


Figura B.7 • Las esquinas de un mapa K son lógicamente adyacentes.

Términos irrelevantes

Considérese los mapas K de la figura B.8. Las entradas *d* representan términos irrelevantes (*don't care*), los que pueden tratarse como ceros o unos según convenga.* Un término irrelevante representa una condición que no puede presentarse durante el funcionamiento. Por ejemplo, si $X = 1$ representa la condición en la cual un ascensor se encuentra en la planta baja mientras que $Y = 1$ representa la condición en la que el ascensor se encuentra en el piso superior, X e Y no podrán ser 1 en el mismo momento, si bien pueden ser ambas 0 simultáneamente. Por lo tanto, en la tabla de verdad que represente el funcionamiento del ascensor, la entrada correspondiente a $X = Y = 1$ podría indicarse como un término irrelevante.

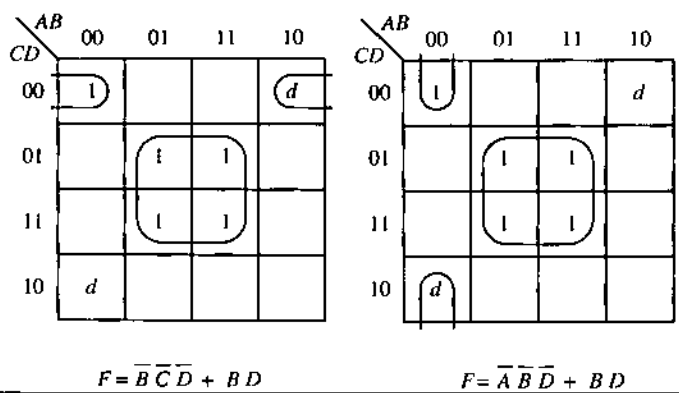


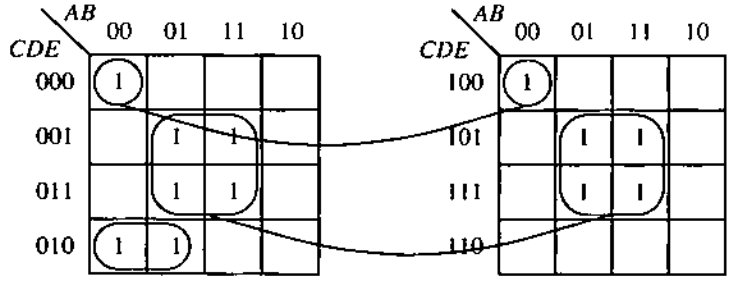
Figura B.8 • Dos ecuaciones mínimas diferentes producidas desde el mismo mapa K.

* *N. de T.*: La denominación "*don't care*", utilizada en el original inglés, suele traducirse como "término redundante". La denominación "término irrelevante" debería dejar más claro el concepto: combinaciones de entrada para las cuales, por distintas razones, no interesa la salida que pueda entregar la función lógica.

En la figura B.8 se muestra una función más compleja en la que se obtienen dos resultados diferentes a partir del mismo procedimiento de minimización. El mapa K de la izquierda considera al término irrelevante de la parte superior derecha como un 1 y al ubicado en la parte inferior izquierda como un 0. El mapa K de la derecha, por el contrario, trata como 0 al término ubicado en la parte superior derecha y como 1 al de la parte inferior izquierda. Ambos mapas K dan por resultado funciones simplificadas del mismo tamaño, y así es posible tener más de una expresión mínima de una misma función booleana. En la práctica, una ecuación puede preferirse sobre la otra, posiblemente con el objeto de reducir el *fan out* de alguna de las variables, o para aprovechar el compartir términos mínimos con otras funciones.

Mapas multidimensionales

La figura B.9 ilustra un mapa K de cinco variables. Cada una de las celdas es adyacente de otras cinco, y, para mantener la propiedad inversa de las celdas adyacentes, el mapa de la izquierda se superpone al de la derecha, creando una estructura tridimensional. Las agrupaciones se hacen ahora en tres dimensiones, como lo muestra la figura. Dado que la estructura tridimensional se ha alojado en una página bidimensional, el lector deberá resolver parte de la visualización del proceso.

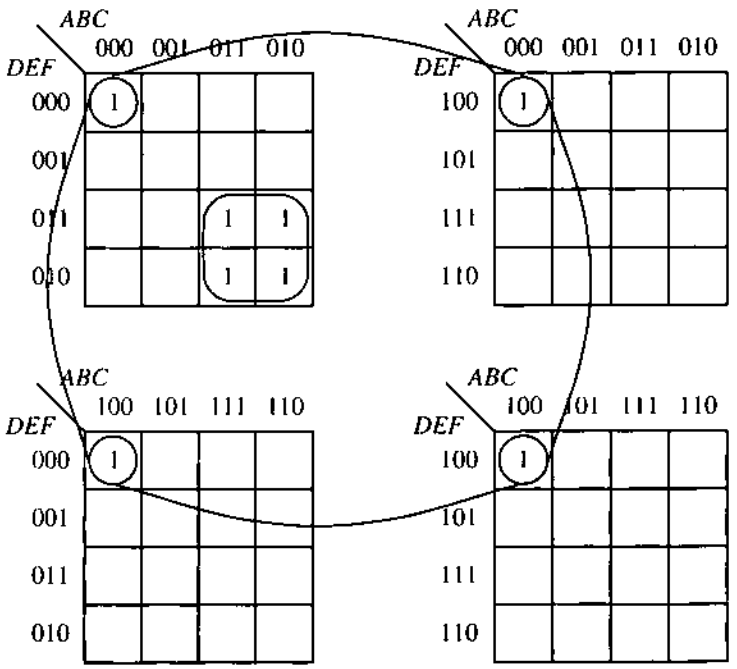


$$F = \overline{A}\,\overline{C}\,D\overline{E} + \overline{A}\,B\,\overline{D}\,\overline{E} + B\,E$$

Figura B.9 • Un mapa K de cinco variables.

En la figura B.10 se ilustra un mapa K de seis variables, en el que los mapas se superponen en cuatro niveles, en el orden arriba-izquierda, arriba-derecha, abajo-derecha y abajo-izquierda. Puede extenderse el uso de mapas K a dimensiones aun mayores para siete o más variables, pero el proceso tiende a ser tedioso, dominado por la dificultad de visualizarlo. La sección B.2.3 describe un proceso algorítmico para cuatro o más variables que conduce a una implementación computacional simple.*

* *N. de T.:* Si se mantiene el planteo de ordenar filas y columnas en la forma de un código Gray, los mapas K de cinco y seis variables pueden adoptar la forma de gráficos únicos, sin necesidad de pensar en tres o más dimensiones. Solo habrá que replantear el concepto de adyacencia.



$$G = \overline{B}\,\overline{C}\,\overline{E}\,\overline{F} + \overline{A}\,B\,\overline{D}\,E$$

Figura B.10 • Un mapa K de seis variables.

Circuitos multinivel

Es importante enfatizar que un mapa K reduce el tamaño de una expresión en dos niveles, medida esta reducción en función de cantidad y tamaño de los términos que la componen. Este proceso no necesariamente produce una forma mínima para circuitos de más de dos niveles. Por ejemplo, la ecuación B.14 se encuentra expresada en su forma mínima de dos niveles, dado que solo se utilizan dos niveles lógicos en su implementación: tres productos lógicos de conjuntos de variables, los que se suman en una compuerta O. El diagrama lógico correspondiente, que se muestra en la figura B.5. tiene un total de 9 entradas de compuertas. Si se factorea algebraicamente una de las variables de la función, puede obtenerse una forma de la función en tres niveles lógicos, de acuerdo con lo que se observa en la ecuación B.15.

$$M = BC + A(B+C) \tag{B.15}$$

El diagrama lógico correspondiente, que se muestra en la figura B.11, tiene 8 entradas a compuertas, lo que significa que se obtiene un circuito más simple si se plantea una solución de tres niveles. Es cierto que ahora surge un mayor retardo entre las entradas y las salidas, por lo

que puede plantearse un nuevo parámetro para medir la complejidad de un circuito: el **retardo** del mismo medido como **número de compuertas**. Un circuito de dos niveles tiene un retardo de dos compuertas, debido a que el camino más largo entre una entrada y una salida tiene solo dos compuertas. El circuito de la figura B.11 tiene un retardo de tres compuertas.

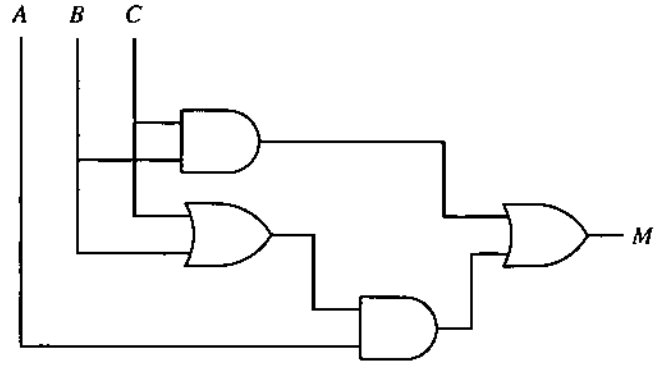
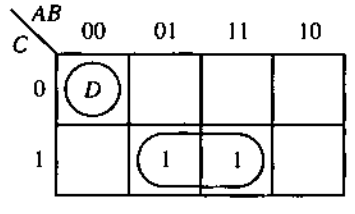


Figura B.11 • La función mayoría implementada en tres niveles con ocho entradas de compuertas.

Si bien existen técnicas que ayudan al diseñador de circuitos a encontrar las soluciones de compromiso entre la cantidad de niveles del circuito y la cantidad de entradas del mismo, el desarrollo de algoritmos que cubran todo el espectro de las alternativas posibles en un tiempo razonable es hoy un problema solo parcialmente resuelto.

Ingreso de variables en el mapa

Se obtiene una forma simplificada de representar una función en un mapa K si se permite la incorporación de variables en las diferentes celdas. Por ejemplo, considérese el mapa K de cuatro variables de la figura B.12. Solo se usan ocho celdas aun cuando hay cuatro variables, las que normalmente requerirían 16 celdas. La **variable ingresada** D se trata como un 1 a los efectos del agrupamiento, el que en este caso da por resultado un grupo de un solo término debido a que no hay unos adyacentes a la celda en que figura la D. La ecuación resultante se indica en la misma figura. Nótese que la variable D aparece en el término $\overline{A}.\overline{B}.\overline{C}.D$, dado que D puede adoptar el valor 0 o 1, aun cuando se la ha considerado como 1 para formar el grupo individual.



$$F = BC + \overline{A}\,\overline{B}\,\overline{C}D$$

Figura B.12 • Ejemplo de un mapa K en el que se ingresa la variable D.

El procedimiento general para llegar a una expresión reducida a partir de un mapa K en el que se han ingresado variables es obtener primero la expresión de las celdas individuales, tratando mientras tanto a las variables ingresadas como 0. Luego, se agregan los términos mínimos para cada variable, tratando los unos como irrelevantes, dado que los mismos ya han sido cubiertos. El proceso sigue hasta que se cubren todas las variables.

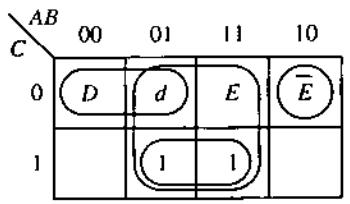


Figura B.13 • Ejemplo de un mapa K en el que se ingresaron las variables D y E.

Considérese el mapa de la figura B.13, en la que D, E y $\bar{E}$ son variables introducidas en el mapa, y d representa un término irrelevante. Si se consideran primero los unos, se obtiene el término BC. Se considera luego la variable D, la que produce el término $\bar{A}\bar{C}D$. Al considerar la variable $\bar{E}$ se obtiene el término BE. Finalmente, considerando la variable E, se obtiene el término $A\bar{B}\bar{C}\bar{E}$. Nótese que una variable ingresada y su complemento se consideran en forma separada, tal como se ha hecho en el ejemplo con E. La ecuación B.16 muestra la forma reducida

$$F = BC + \bar{A}\bar{C}D + BE + A\bar{B}\bar{C}\bar{E} \tag{B.16}$$

B.2.3 El método tabular

Tanto en los casos de funciones de una salida como en los de múltiples salidas, se suele utilizar un procedimiento automatizado para la reducción de expresiones booleanas. El método tabular, conocido como método de **Quine-McCluskey,** obtiene sucesivos productos booleanos cruzados entre grupos de términos que difieren en una sola variable, y luego utiliza el menor de los conjuntos de términos reducidos que cubra toda la función. El proceso es más sencillo de implementar en computadora que el método gráfico; además, una extensión de este método permite compartir términos entre funciones.

Reducción de funciones de una salida

La tabla de verdad de la figura B.14 representa una función F de cuatro variables A, B, C y D, e incluye tres términos irrelevantes. El proceso de reducción tabular comienza agrupando los términos mínimos para los cuales F es cierta, de acuerdo con la cantidad de unos que incluye la expresión binaria de cada término mínimo. En este proceso, las combinaciones redundantes se deben considerar como unos. El término mínimo 0000 no contiene unos, por lo que forma un grupo unitario, de acuerdo con la figura B.15a. Los tér-

minos 0001, 0010, 0100 y 1000 contienen todos un único uno, pero dado que la función es cierta solo para el término mínimo 0001, este único término forma un grupo nuevo.

A	*B*	*C*	*D*	*F*
0	0	0	0	*d*
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	*d*
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	*d*

Figura B.14 • Una tabla de verdad que representa una función con términos irrelevantes.

El grupo siguiente tiene dos unos en cada término mínimo, y existen seis términos mínimos que podrían pertenecer al mismo. Solo los términos mínimos 0011, 0101, 0110 y 1010 tienen asignados un 1 como valor de la función, por lo que son esos los que forman el segundo grupo. Existen tres entradas ciertas en el próximo grupo, que corresponde a los elementos que se escriben con tres unos. Los términos para los que la función es cierta son

Configuración inicial

A	*B*	*C*	*D*	
0	0	0	0	√
0	0	0	1	√
0	0	1	1	√
0	1	0	1	√
0	1	1	0	√
1	0	1	0	√
0	1	1	1	√
1	0	1	1	√
1	1	0	1	√
1	1	1	1	√

(a)

Luego de la primera simplificación

A	*B*	*C*	*D*	
0	0	0	x	*
0	0	x	1	√
0	x	0	1	√
0	x	1	1	√
x	0	1	1	√
0	1	x	1	√
x	1	0	1	√
0	1	1	x	*
1	0	1	x	*
x	1	1	1	√
1	x	1	1	√
1	1	x	1	√

Luego de la segunda simplificación

A	*B*	*C*	*D*	
0	x	x	1	*
x	x	1	1	*
x	1	x	1	*

(c)

(b)

Figura B.15 • El proceso de simplificación tabular.

0111, 1011 y 1110. Finalmente, hay un único término cierto formado por cuatro unos, por lo que ese término mínimo constituye el último grupo. Para tablas de verdad más grandes, el proceso continúa hasta que se hayan completado todas las combinaciones que hacen cierta la función. Los grupos se organizan de modo tal que los grupos adyacentes difieren en 1 en la cantidad de unos que los expresan, tal como se observa en la figura B.15a.
El próximo paso es formar un consenso (forma lógica de un producto cruzado) entre cada par de grupos adyacentes para todos los términos que difieren en una variable. Se rescata del apéndice A la forma general del teorema del consenso:

$$XY + \bar{X}Z + YZ = XY + \bar{X}Z \tag{B.17}$$

El término YZ es redundante, ya que se encuentra cubierto por los términos restantes, por lo tanto, puede eliminarse. Este teorema se demuestra algebraicamente en la siguiente forma:

$$\begin{aligned} XY + \bar{X}Z + YZ &= XY + \bar{X}Z(X + \bar{X}) \\ &= XY + \bar{X}Z + XYZ + \bar{X}YZ \\ &= XY + XYZ + \bar{X}Z + \bar{X}YZ \\ &= XY(1 + Z) + \bar{X}Z\,(1 + Y) \\ &= XY + \bar{X}Z \end{aligned}$$

El teorema del consenso ofrece una forma dual:

$$(X + Y)(\bar{X} + Z)(Y + Z) = (X + Y)(\bar{X} + Z) \tag{B.18}$$

La idea de aplicar el consenso en la reducción tabular tiene que ver con el aprovechamiento de la propiedad inversa del álgebra de Boole, en forma similar a lo realizado con el método de los mapas K en la sección anterior. Por ejemplo, 0000 y 0001 difieren solo en la variable D, por lo que en la lista de la figura B.15b se indica esta diferencia escribiendo 000x. La letra x indica la posición de la variable eliminada, que en este caso es D. Los términos mínimos 0000 y 0001 se tildan para indicar que han sido cubiertos en la tabla reducida.

Luego de comparar cada término del primer grupo con cada término del segundo grupo, se avanza con el objeto de lograr reducciones entre los términos del segundo y tercer grupo. Nótese que existe la posibilidad de que algunos términos no puedan combinarse en otros menores por diferir en más de una variable. Por ejemplo, los términos 0001 y 0011 se combinan dando por resultado el término 00x1, como se ve al comienzo del segundo grupo de la figura B.15b, pero los términos 0001 y 0110 no pueden combinarse por cuanto difieren en tres variables.

Una vez que un término ha sido tildado, puede seguir siendo usado en el proceso de reducción por aplicación de la propiedad de idempotencia. El objetivo de este paso del proceso es descubrir todos los posibles términos reducidos para encontrar, en un paso posterior, el menor conjunto de términos con el que pueda cubrirse la función.

El proceso continúa para todos los grupos remanentes. Cada término que no haya quedado cubierto luego de las agrupaciones por consenso se identifica con un asterisco para indicar que el mismo es un implicante primo. Luego de haber realizado la primera reducción de este ejemplo, todos los términos mínimos de la figura B.15a han quedado cubiertos, por lo que, al momento, no existen implicantes primos.

Luego de realizada la primera reducción, se puede iniciar una nueva. Para poder combinar ahora dos términos ya reducidos, estos deben diferir nuevamente en una sola variable. Los indicadores tienen que estar ubicados en la misma posición y solo una de las demás variables debe diferir. La primera entrada de la figura B.15b tiene la x en la columna de la derecha, lo que no coincide con ninguno de los elementos del segundo grupo, por lo que se lo identifica con un asterisco para indicar que no puede volver a ser agrupado y, por lo tanto, pasa a ser un implicante primo de la función. Si ahora se analizan el segundo y tercer grupo de la figura B.15b, los términos 00x1 y 01x1 se combinan para obtener la forma más simplificada 0xx1, indicada en la figura B.15c. El proceso continúa hasta que se completa la segunda simplificación, que se muestra en la figura B.15c.

Tomados como un ente completo, los implicantes primos forman un conjunto que cubre íntegramente la función, si bien no es necesario que dicho conjunto sea mínimo. Con el objeto de obtener una cobertura mínima, se construye una **tabla de opciones**, la que se representa en la figura B.16. Se asigna un renglón de la tabla a cada implicante primo. Las columnas representan los términos mínimos de la función original que deben ser cubiertos por la función simplificada. Los términos irrelevantes no necesitan ser cubiertos y, por lo tanto, no se incluyen en la lista.

Implicantes primos	Términos mínimos						
	0001	0011	0101	0110	0111	1010	1101
000x	√						
*011x				√	√		
*101x						√	
0xx1	√	√	√		√		
xx11		√			√		
*x1x1			√		√		√

Figura B.16 • Tabla de opciones.

Se tilda cada uno de los casilleros correspondientes a un implicante primo que cubre a un término mínimo. Por ejemplo, dado que el implicante primo 000x cubre al término mínimo 0001, se tilda la casilla correspondiente a dicha intersección. Algunos implicantes primos cubren varios términos mínimos, como en el caso del 0xx1, que cubre cuatro términos mínimos. Luego de haber considerado todas las casillas, se identifican aquellas

columnas que contienen un solo tilde. Un único tilde en una columna implica que ese término mínimo está cubierto por un único implicante primo, el que se identifica con un asterisco para indicar que se trata de un implicante primo **esencial**.

Los implicantes primos esenciales no pueden eliminarse y deben estar incluidos en la ecuación reducida que represente a la función. En este ejemplo, los implicantes primos 011x, 101x y x1x1 son esenciales. Un implicante primo esencial puede cubrir más de un término mínimo, por consiguiente, se genera una **tabla reducida de opciones** en la que se eliminan los implicantes primos esenciales y los términos mínimos ya cubiertos por ellos, la que se presenta en la figura B.17. La tabla reducida puede volver a contener nuevos implicantes primos esenciales, en cuyo caso habrá que crear una segunda tabla reducida de opciones, continuándose el proceso hasta que la tabla final contenga solo implicantes primos no esenciales.

Conjunto de combinaciones	Términos mínimos	
	0001	0011
X 000x	√	
Y 0xx1	√	√
Z xx11		√

Conjunto 1 **Conjunto 2**
000x 0xx1
xx11

Figura B.17 • Tabla reducida de opciones.

Los implicantes primos que aún se mantienen en la tabla reducida de opciones forman el llamado **conjunto elegible**, a partir del cual se obtendrá el conjunto mínimo de implicantes que permita cubrir el total de los términos mínimos aun remanentes. Como se observa en la figura B.17, existen dos conjuntos de implicantes primos que cubren los dos términos mínimos pendientes. Dado que el segundo conjunto tiene menos elementos que el primero, se opta por seleccionar ese conjunto, obteniéndose así la ecuación mínima representativa de *F*, formada por los implicantes primos esenciales y por los implicantes elegibles contenidos en el segundo conjunto.

$$F = \bar{A}BC + A\bar{B}C + BD + \bar{A}D \tag{B.19}$$

En lugar de obtener el conjunto de implicantes elegibles por inspección visual, se puede desarrollar un procedimiento algebraico con tal objetivo. El proceso se inicia asignando una variable a cada uno de los implicantes del conjunto elegible, como se muestra en la figura B.17. Se describe cada columna de la tabla de opciones reducida a través de una expresión lógica, como las que se indican a continuación:

Columna	Suma lógica
0001	$(X+Y)$
0011	$(Y+Z)$

Con el objeto de encontrar un conjunto que cubra la función completamente, se agrupan los implicantes primos de modo que haya al menos un tilde en cada columna. Esto significa que debe cumplirse la siguiente relación, en la que G representa los términos de la tabla reducida de opciones:

$$G = (X+Y)(Y+Z)$$

Si se aplican las propiedades del álgebra de Boole, se obtiene:

$$G = (X+Y)(Y+Z) = XY + XZ + Y + YZ = XZ + Y$$

Cada uno de los términos de la ecuación representa un conjunto de implicantes primos que cubre los términos de la tabla reducida de opciones. El término (Y) representa el menor conjunto de implicantes primos (0xx1) que cubre los términos remanentes. Se obtiene la misma expresión final:

$$F = \bar{A}BC + A\bar{B}C + BD + \bar{A}D \tag{B.20}$$

Reducción de funciones múltiples

El método de reducción tabular simplifica una única función booleana. Cuando se tiene más de una función lógica que utiliza las mismas variables, puede ser factible compartir términos, con lo que se obtiene, en el conjunto, una expresión más reducida de las ecuaciones. El método que aquí se describe forma la intersección entre todas las posibles combinaciones de términos compartidos y selecciona el conjunto mínimo requerido para cubrir todas las funciones.

Como ejemplo, considérese la tabla de verdad de la figura B.18, que representa tres funciones de tres variables. La notación m_i define los términos mínimos de acuerdo con los subíndices que se indican en la misma tabla.

Término mínimo	A	B	C	F_0	F_1	F_2
m_0	0	0	0	1	0	0
m_1	0	0	1	0	1	0
m_2	0	1	0	0	0	1
m_3	0	1	1	1	1	1
m_4	1	0	0	0	1	0
m_5	1	0	1	0	0	0
m_6	1	1	0	0	1	1
m_7	1	1	1	1	1	1

Figura B.18 • Tabla de verdad de tres funciones de tres variables cada una.

La forma canónica (no reducida) de las ecuaciones booleanas es

$$F_0(A, B, C) = m_0 + m_3 + m_7$$
$$F_1(A, B, C) = m_1 + m_3 + m_4 + m_6 + m_7$$
$$F_2(A, B, C) = m_2 + m_3 + m_6 + m_7$$

Para cada una de las combinaciones de funciones se plantea una intersección como las que se indican:

$$F_{0,1}(A, B, C) = m_3 + m_7$$
$$F_{0,2}(A, B, C) = m_3 + m_7$$
$$F_{1,2}(A, B, C) = m_3 + m_6 + m_7$$
$$F_{0,1,2}(A, B, C) = m_3 + m_7$$

Si se utiliza el método tabular de reducción descripto en la sección anterior, se obtienen los siguientes implicantes primos:

Función	Implicante primo
F_0	000, x11
F_1	0x1, 1x0, x11, 11x
F_2	x1x
$F_{0,1}$	x11
$F_{0,2}$	x11
$F_{1,2}$	x11,11x
$F_{0,1,2}$	x11

La lista de implicantes primos se reduce eliminando en cada función a aquellos implicantes primos que aparezcan en funciones de mayor orden. Por ejemplo, x11 aparece en $F_{0,1,2}$, por lo que no requiere ser incluido en las restantes funciones. En forma similar, 11x aparece en $F_{1,2}$, por lo que no se requiere que aparezca ni en F_1 ni en F_2. (En este caso, ni siquiera aparece como implicante de F_2.) Siguiendo con el procedimiento, se obtiene un conjunto reducido de implicantes primos:

Función	Implicante primo
F_0	000
F_1	0x1, 1x0
F_2	x1x
$F_{0,1}$	ninguno
$F_{0,2}$	ninguno
$F_{1,2}$	11x
$F_{0,1,2}$	x11

Con estos elementos se construye una tabla de opciones para las múltiples salidas, como la que se muestra en la figura B.19. Las filas corresponden a los implicantes primos y las columnas, a los términos mínimos que deben cubrirse en cada función. Se anulan aquellos sectores de las filas en las que los implicantes de una función no pueden usarse para cubrir alguna otra función. Por ejemplo, el implicante primo 000 se obtiene desde la función F_0, y dado que no puede usarse para cubrir término alguno de las funciones F_1 o F_2, dichas zonas aparecen anuladas. Si de hecho algún implicante correspondiente a F_0 pudiera usarse para cubrir algún término mínimo de alguna de las funciones restantes, debería aparecer en algunas de las funciones planteadas como intersección, por ejemplo en $F_{0,1}$ o $F_{0,1,2}$.

Términos mínimos / Implicantes primos		$F_0(A,B,C)$			$F_1(A,B,C)$					$F_2(A,B,C)$			
		m_0	m_3	m_7	m_1	m_3	m_4	m_6	m_7	m_2	m_3	m_6	m_7
F_0	*000	√											
F_1	*0x1				√	√							
F_1	*1x0						√	√					
F_2	*x1x									√	√	√	√
$F_{1,2}$	11x							√	√			√	√
$F_{0,1,2}$	*x11		√	√		√			√		√		√

Figura B.19 • Tabla de opciones para funciones de múltiples salidas.

La forma mínima de las ecuaciones de salida se obtiene de modo semejante al del proceso tabular de simplificación. Se inicia el proceso buscando todos los implicantes primos esenciales. Por ejemplo, el término mínimo m_0 de la función F_0 solo está cubierto por el implicante primo 000, por lo que 000 es un implicante primo esencial. La fila que contiene a 000 se elimina de la tabla, junto con todas las columnas tildadas en esa fila. El proceso continúa ya sea hasta que todas las funciones hayan quedado cubiertas o hasta que solo queden los implicantes primos no esenciales, momento en el cual se seleccionará el conjunto mínimo de implicantes primos no esenciales necesario para cubrir las funciones remanentes, utilizando el método descripto en la sección anterior.

Los implicantes primos esenciales son los indicados con asteriscos en la figura B.19. Para este caso, solo queda un implicante primo no esencial (11x), pero dado que todos los términos mínimos están cubiertos por los implicantes primos esenciales, no hay necesidad de construir una tabla reducida. Las ecuaciones finales correspondientes son:

$$F_0(A, B, C) = \bar{A}\bar{B}\bar{C} + BC$$

$$F_1(A, B, C) = AC + A\bar{C} + BC$$

$$F_2(A, B, C) = B$$

B.2.4 Simplificación lógica: su efecto sobre la velocidad y la eficiencia

Hasta este punto, las características físicas que afectan a la eficiencia han sido prácticamente ignoradas, habiéndose enfocado el análisis enteramente sobre aspectos de organización tales como la cantidad de niveles y de compuertas de los circuitos. En esta sección se analizan algunas consideraciones prácticas referidas a los circuitos digitales.

Velocidad de conmutación

El tiempo de propagación (latencia) entre entradas y salidas de una compuerta lógica es un efecto continuo, aun cuando en las primeras secciones del apéndice A se consideró despreciable el tiempo de propagación. Los cambios en las entradas de una compuerta lógica también son un efecto continuo. En la figura B.20, la entrada de la compuerta NO tiene un tiempo de transición finito, el que se mide como el tiempo requerido para que la señal pase desde el 10% al 90% de su valor final. Estos tiempos suelen conocerse como tiempo de crecimiento (*rise time*), cuando se trata de una señal creciente, y tiempo de caída (*fall time*), cuando la señal es decreciente.

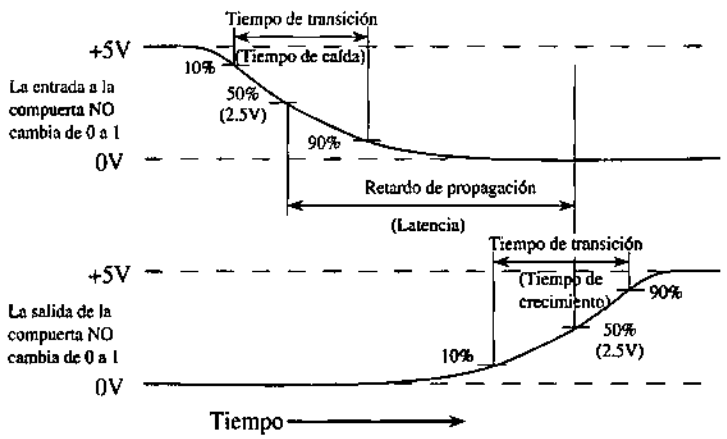


Figura B.20 • Tiempo de propagación de una compuerta NO (adaptado de Hamacher y otros).

El tiempo de propagación es el tiempo que transcurre entre el 50% del valor de la señal de entrada y el 50% del valor de la señal de salida. El tiempo de propagación sufre la influencia de una cantidad de parámetros, siendo la potencia eléctrica uno de los parámetros sobre los que se dispone de cierto grado de control. El tiempo de propagación disminuye, hasta un cierto límite, a medida que crece el consumo de energía de alimentación. Una regla no escrita indica que el producto de la potencia consumida por el tiempo de propagación debe permanecer aproximadamente constante. Si bien generalmente se busca que la lógica sea veloz, no se pretende pagar esa velocidad con una alta disipación de potencia dado que la potencia consumida se muestra

a su vez como calor que debe eliminarse para lograr condiciones de operación confiables y seguras.*

En la familia lógica CMOS (transistores de efecto de campo, complementarios), la disipación de potencia crece con la velocidad. A una frecuencia de conmutación de 1 Mhz, la disipación de potencia de una compuerta CMOS es de alrededor de 1 mW. A estos niveles de disipación, 10.000 compuertas lógicas disipan una potencia total en el orden de los 10 Watt, valor que está cerca del límite de disipación de calor que puede manejar un circuito integrado único de 1 cm^2 mediante técnicas convencionales.

Sin embargo, los circuitos integrados de tecnología CMOS pueden tener alrededor de 10^7 compuertas lógicas y operar a frecuencias de algunos cientos de Mhz. Estas cantidades de compuertas y estas velocidades se obtienen parcialmente mediante el aumento de la superficie del circuito integrado, aun cuando lo que se logra influye en un factor algo mayor que 10. La clave para lograr una buena cantidad de componentes y velocidad de propagación, sin incrementar apreciablemente la potencia disipada, consiste en hacer conmutar solo una fracción de las compuertas lógicas en cada instante de tiempo, lo que, afortunadamente, coincide con el modo de operación típico de un circuito integrado.

Cantidad de niveles lógicos de un circuito

El tiempo de propagación entre las entradas y salidas de un circuito está determinado por la cantidad de compuertas lógicas incluidas en el camino más largo que pueda existir entre cualquier entrada y cualquier salida. Esta cantidad de compuertas determina la cantidad de niveles o **profundidad** del circuito. En general, un circuito con pocos niveles opera a mayor velocidad que aquel circuito que tiene mayor cantidad de niveles de compuertas. Existen distintas formas de disminuir la cantidad de niveles de un circuito, las que en general implican el aumento de algún otro parámetro del mismo. A continuación, se analiza una de estas soluciones de compromiso.

En el apéndice A se utilizó un multiplexor para implementar la función mayoría. Considérese la utilización de un multiplexor de cuatro entradas de control para implementar la ecuación B.21. Esta ecuación se encuentra expresada en dos niveles, dado que solo se utilizan dos niveles de compuertas en su representación: seis productos lógicos que llevan a una compuerta O. Esta función puede implementarse con un único multiplexor, el que se muestra a la izquierda de la figura B.21. La profundidad circuital correspondiente es dos (esto es, la configuración de compuertas internas del multiplexor tiene dos retardos). Si se opera sobre la expresión despejando A y B, se obtiene una función en cuatro niveles representada en la ecuación B.22, ecuación que se representa con el circuito lógico de la figura B.21, derecha.

* *N. de T.*: En la práctica, el producto mencionado se conoce como "factor de mérito". La actualización tecnológica permite obtener circuitos lógicos que mejoran la velocidad sin aumentar proporcionalmente la potencia eléctrica, lo que implica un factor de mérito menor. Un factor de mérito menor implica una tecnología más eficiente.

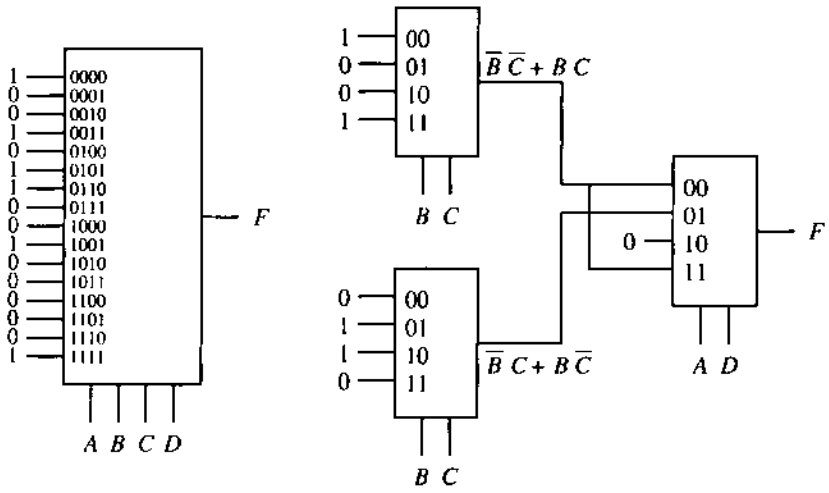


Figura B.21 • Una función de cuatro variables implementada con un multiplexor de 16 entradas y con multiplexores de 4 entradas.

$$F(A, B, C, D) = \overline{A}\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,\overline{B}CD + \overline{A}B\overline{C}D + A\overline{B}\,\overline{C}D + \overline{A}\,\overline{B}CD + ABCD \qquad \text{(B.21)}$$

$$F(A, B, C, D) = \overline{A}\,\overline{B}(\overline{C}\,\overline{D} + CD) + \overline{A}B(\overline{C}D + C\overline{D}) + A\overline{B}(\overline{C}D) + AB(\overline{C}D) \qquad \text{(B.22)}$$

La cantidad de entradas de un multiplexor de cuatro entradas de datos es 18, según surge de la figura A.23, incluyendo los inversores, de modo que la cantidad de entradas del circuito con tres multiplexores es de 18 x 3 = 54. Un único multiplexor de 16 entradas de datos tiene 100 entradas en sus compuertas. La implementación con multiplexores de cuatro entradas de datos tiene cuatro niveles lógicos (sin contar inversores), en tanto que la implementación con un único multiplexor de 16 entradas de datos tiene dos niveles lógicos. Se ha logrado reducir la complejidad total del circuito a costa de un aumento de su cantidad de niveles.

Si bien existen técnicas que colaboran con el diseñador de circuitos para permitirle encontrar soluciones de compromiso entre la complejidad de un circuito y la cantidad de niveles lógicos del mismo, el desarrollo de algoritmos que cubran todas las posibles alternativas en un tiempo razonable es un problema solucionado solo parcialmente.

Cantidad de entradas versus profundidad circuital

Supóngase que surja la necesidad de utilizar una compuerta O como la de la figura A.23, pero que solo se encuentren disponibles compuertas O de dos entradas. ¿Qué debe hacerse? Este problema, de ocurrencia habitual, surge en muchas situaciones durante un diseño. La propiedad asociativa del álgebra de Boole se puede utilizar para descomponer la compuerta O que tiene cuatro entradas en una configuración de compuertas O de dos entradas cada una, tal como se ilustra en la figura B.22. En gene-

ral, la descomposición de la compuerta O de cuatro entradas debería realizarse en la forma de un árbol balanceado para reducir la cantidad de niveles lógicos. Puede utilizarse también un árbol desbalanceado, como el que se ilustra en la misma figura B.22, obteniéndose un circuito funcionalmente equivalente, con la misma cantidad de compuertas lógicas que en el caso del circuito balanceado pero con mayor cantidad de niveles lógicos.

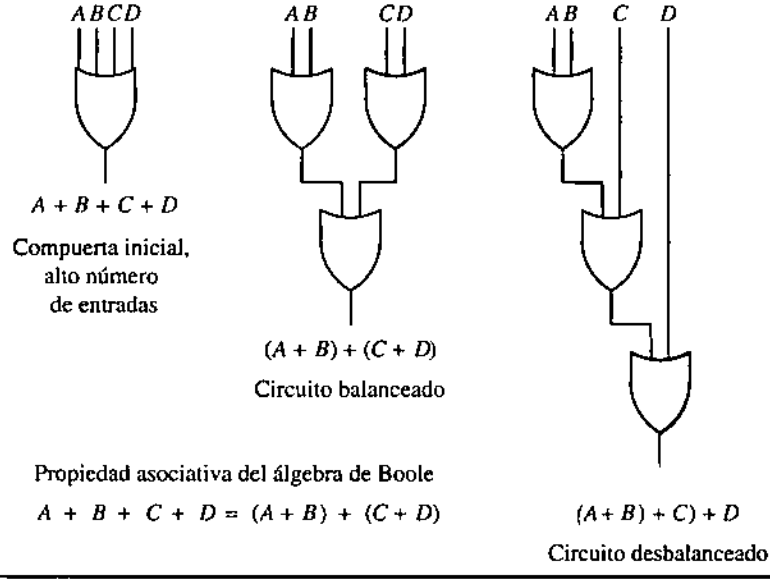


Figura B.22 • Una compuerta lógica con cuatro entradas y su reemplazo por configuraciones lógicamente equivalentes con compuertas de dos entradas.

Si bien es importante reducir la profundidad de un circuito con el objeto de bajar los tiempos de propagación entre entradas y salidas, una razón que puede hacer optar por el árbol desbalanceado es que este posee una mínima cantidad de compuertas en cada uno de los niveles, lo que puede hacer que sea simple partir el árbol en partes físicamente ubicadas en distintos circuitos integrados. Esto refleja una situación práctica que suele encontrarse en el encapsulado de circuitos digitales. Cuando una compuerta de N entradas se convierte en un mapa de compuertas lógicas de F entradas, el número de niveles del árbol balanceado es $[log_F(N)]$, en tanto que, para una conversión similar, el número de niveles del árbol desbalanceado es de $(N - 1)/(F - 1)$ compuertas.

En teoría, cualquier función binaria puede realizarse con dos niveles de compuertas lógicas si se considera una gran cantidad de compuertas Y seguidas de una gran cantidad de compuertas O, teniendo ambas compuertas niveles arbitrariamente grandes de entradas y de salidas. Por ejemplo, cualquier programa de computadora podría ser compilado en solo dos niveles de compuertas si se lo presentara en paralelo a la entrada de un circuito booleano con un nivel de compuertas Y seguido por una compuerta O, destinadas a implementar la función. No obstante, tal circuito sería extremadamente

costoso porque debería tener en cuenta todas y cada una de las combinaciones booleanas posibles.

En la mayor parte de las familias lógicas se hace muy costosa la implementación de capacidades de carga mayores que 10, debido a la pérdida de eficiencia. No obstante, se sigue el álgebra de Boole en expresiones de dos niveles para describir circuitos digitales complejos y con grandes capacidades de salida. Las expresiones booleanas de dos niveles se convierten luego en expresiones multinivel, de acuerdo con las limitaciones de entradas y salidas de la tecnología utilizada. Se dice que los valores óptimos de las características de *fan in* y *fan out* deben valer, aproximadamente, $e = 2{,}7$ (véase la obra de C. Mead y L. Conway), en función de los tamaños de los transistores requeridos para llevar una señal desde un circuito integrado hasta la pata del encapsulado del mismo. El resultado obtenido surge de analizar las capacitancias de los conexionados, los tiempos de crecimiento de las señales y otras consideraciones. El resultado no puede aplicarse a todos los aspectos del cálculo dado que no toma en cuenta la eficiencia total, que puede producir variaciones locales que modifiquen apreciablemente la regla del número *e*. Los circuitos electrónicos digitales tienen, habitualmente, características de entrada y salida en el rango que va de 2 a 10.

B.3 Reducción de estados

En el apéndice A se analizó un método para el diseño de máquinas de estados finitos sin considerar que podría existir una máquina funcionalmente equivalente pero con menor número de estados. En esta sección se analizarán los métodos que permitan reducir la cantidad de estados. Se inicia el análisis con la descripción de una máquina de estados finitos que tiene determinada cantidad de estados, y se plantea luego la hipótesis de encontrar una máquina funcionalmente equivalente que contenga un único estado. Se aplican todas las combinaciones de entradas a la máquina hipotética y se observan las salidas. Si para la misma combinación de valores de entrada, producidos en distintos instantes de tiempo, la máquina de estados finitos entrega salidas diferentes, existen al menos dos estados distinguibles y, por ende, no equivalentes. Los estados distinguibles se colocan en grupos separados y el proceso continúa hasta que no pueda hacerse distinción adicional alguna. Si en alguno de los grupos remanentes existe más de un estado, los mismos son equivalentes, por lo que puede obtenerse una máquina equivalente de menor tamaño, en la que todos los estados de cada uno de los grupos pueden reemplazarse por un solo estado por grupo.

Como ejemplo, considérese la máquina M_0 descripta por la tabla de estados de la figura B.23. El proceso de reducción comienza partiendo de suponer que los cinco estados pueden reducirse a uno solo, obteniéndose así la partición P_0 para una nueva máquina M_1:

$$P_0 = (ABCDE)$$

Estado actual \ Entrada	X = 0	X = 1
A	C/0	E/1
B	D/0	E/1
C	C/1	B/0
D	C/1	A/0
E	A/0	C/1

Figura B.23 • Descripción de la máquina de estados M_0 a reducir.

Si se aplica una entrada unitaria a la máquina original M_0 y se observa la salida, se puede ver que cuando M_0 está en su estado A, y se le aplica una entrada de valor 0, la salida es 0. Cuando la máquina está en su estado A y se le aplica un 1, la salida es 1. Los estados B y E se comportan en forma similar, pero los estados C y D producen salidas de 1 y 0 cuando las entradas son 0 y 1, respectivamente. Por lo tanto, se puede ver que los estados A, B y E se distinguen de los estados C y D, por lo que se obtiene una nueva partición P_1:

$$P_1 = (ABE)(CD)$$

Luego de aplicar a M_0 una única entrada, la máquina se encontrará en el grupo ABE o en el grupo CD. Se requiere observar la conducta de la máquina a partir de su nuevo estado. Una forma de hacerlo es mediante la enumeración del conjunto de posibles estados futuros en la forma de un árbol, como lo muestra la figura B.24. El proceso de construcción del árbol comienza listando todos los estados de la misma partición. La partición inicial de la máquina M_0 se coloca como raíz del árbol. Luego de aplicar un 0 en la entrada de M_0, el estado siguiente puede ser alguno de los estados C, D, C, C, A según se comience desde A, B, C, D, E, respectivamente. Esto se muestra en la partición $(CDA)(CC)$ que aparece relacionada con la entrada 0, un escalón por debajo de la partición de base. La salida generada por el grupo (CDA) es diferente de la salida producida por el grupo (CC), por lo tanto, los estados iniciales correspondientes a cada grupo son distinguibles. Los estados que se distinguen son los grupos (ABE) y (CD), los que forman, según se muestra, la partición $(ABE)(CD)$.

En forma similar, luego de aplicar un 1 en la entrada de M_0, el estado siguiente será alguno de los estados E, E, B, A o C, en función de los estados iniciales A, B, C, D, E, respectivamente. Esto se observa sobre la derecha del árbol. Para obtener el próximo nivel, se deben analizar en forma separada los grupos (CDA) y (CC). Cuando se aplique un 0 en la entrada estando M_0 en alguno de los estados C, D o A, las salidas serán las mismas para los estados C y D (la salida es 1, siendo los próximos estados C y C), pero serán diferentes para el estado A (la salida es 0 y el próximo estado es C). Esto se observa como $(CC)(C)$ en el trayecto 0,0 a partir de la raíz.

Asimismo, si se aplica un 0 en la entrada de M_0 cuando se encuentra en alguno de los estados C o D, las salidas son las mismas, y el conjunto de estados de destino para el re-

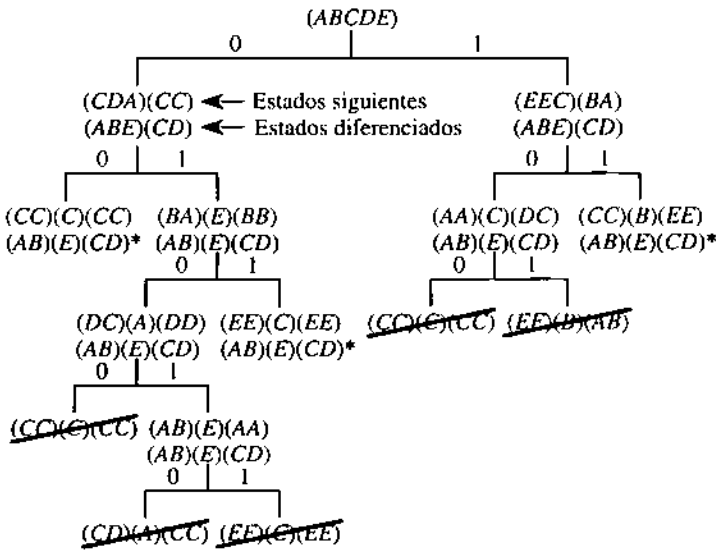


Figura B.24 • Árbol de estados para M_0.

corrido 0,0 a partir de la raíz es $(CC)(C)(CC)$. Esto corresponde, retrocediendo hasta la raíz del árbol, a una partición $(AB)(E)(CD)$ de los estados iniciales. Por consiguiente, en este punto, A no se distingue de B, y C no se distingue de D, pero cada grupo encerrado entre paréntesis se distingue de cualquier otro si se aplica a M_0 la secuencia 0,0 y se analizan las salidas, con independencia del estado inicial.

Continuando de la misma manera, se expande el árbol hasta que no puedan encontrarse particiones mejores. Por ejemplo, cuando una partición contiene un grupo de estados que no pueden distinguirse, como en el caso de $(CC)(C)(CC)$, se coloca un asterisco en posición adyacente a la partición de los estados iniciales correspondientes, dejándose de expandir el árbol a partir de ese punto. El árbol de la figura B.24 ha sido expandido más allá de este punto para ilustrar varias situaciones que podrían surgir.

Si se genera una partición que se volverá a visitar en algún otro lugar del árbol, se la atraviesa con una barra, cesando a partir de ahí la expansión del árbol. Por el hecho de compartir particiones similares, $(CD)(A)(CC)$ se considera igual a $(CD)(A)(C)$, en tanto que $(AA)(C)(CD)$ se considera igual que $(A)(C)(CD)$, que a su vez coincide con $(DC)(A)(C)$ y con $(CD)(A)(C)$. Por lo tanto, las particiones $(CD)(A)(CC)$ y $(AA)(C)(DC)$ se consideran la misma. Luego de construir el árbol, las particiones marcadas con asteriscos representan los estados no distinguibles. Cada grupo de paréntesis en una partición marcada identifica a un grupo de estados no distinguibles. Para la máquina M_0, los estados A y B no se distinguen, así como no se distinguen C y D. Por consiguiente, puede construirse una máquina funcionalmente equivalente a M_0 que contenga solo tres estados. En la misma, A y B se funden en un solo estado, así como C y D también se combinan en un solo estado.

El proceso de construcción del árbol de estados es un proceso laborioso debido a su potencial tamaño, pero se lo ha utilizado aquí para entender un método simple. En vez de

construir todo el árbol, puede observarse simplemente que una vez que se dispone de la primera partición P_1, la partición siguiente puede construirse observando los estados siguientes de cada grupo y notando que si dos estados dentro de un grupo tienen próximos estados que se encuentran en grupos diferentes, son distinguibles debido a que las salidas resultantes serán eventualmente diferentes. Esto puede verse planteando un árbol para los elementos distinguibles. Comenzando con P_1 en M_0 se observa que los estados A y B tienen como próximos estados a C y D para la entrada en 0, y tienen como próximo estado a E para una entrada de valor 1. Por lo tanto, A y B se agrupan en la partición siguiente. El estado E, no obstante, tiene como estados siguientes a A y C cuando las entradas son 0 y 1, respectivamente, lo que difiere de los futuros estados correspondientes a A y B, por lo que E se distingue de los estados A y B. Continuando con el grupo (CD) de P_1, la partición siguiente se obtiene como se indica a continuación. Luego de $P_2 = (AB)(CD)(E)$, si se aplica el método en una nueva iteración, se repite la partición, lo que corresponde a una de las condiciones para detener el proceso:

$$P_3 = (AB)(CD)(E) \surd$$

En este punto no pueden lograrse nuevas distinciones, por lo que la máquina resultante M_1 tiene tres estados en su forma reducida. Si se realiza la asignación $A' = AB$, $B' = CD$ y $C' = E$, en la que los símbolos de la izquierda de cada igualdad representan los estados de la máquina M_1, puede crearse una tabla de estados reducida, la que se muestra en la figura B.25.

Entrada / Estado actual	X	
	0	1
$AB: A'$	$B'/0$	$C'/1$
$CD: B'$	$B'/1$	$A'/0$
$E: C'$	$A'/0$	$B'/1$

Figura B.25 • Tabla de estados reducida para M_1.

B.3.1 El problema de la asignación de estados

Puede ocurrir que, si en una misma máquina se realizan diferentes asignaciones de estados, se obtengan implementaciones diferentes. Por ejemplo, considérese la máquina M_2 que se muestra en la parte izquierda de la figura B.26. Se han planteado dos asignaciones de estados diferentes. La asignación de estados SA_0 consiste en un simple ordenamiento numérico de $A \rightarrow 00$, $B \rightarrow 01$, $C \rightarrow 10$ y $D \rightarrow 11$. La asignación SA_1 difiere de SA_0 en que se han cruzado las asignaciones de C y D. Se considera una implementación de M_2 que utiliza la asignación de estados SA_0, con compuertas Y, O y NO, y se utilizan mapas K pa-

ra reducir los tamaños de las ecuaciones. La figura B.27 muestra el resultado de reducir las funciones S_0 y S_1 correspondientes a los estados futuros y la función de salida Z. El circuito correspondiente es un circuito con 29 entradas, valor que se obtiene contando la cantidad de variables y la cantidad de términos en las ecuaciones (nótese que existe un término que se comparte y, por lo tanto, se computa una sola vez). Si en cambio se utiliza la asignación SA_1, se obtiene un total de 6 entradas, tal como se ve en la figura B.28, (s_0 y s_1 no contribuyen a la cuenta de la cantidad de entradas, dado que no se las ingresa en compuerta alguna).

Entrada / Estado actual	X = 0	X = 1
A	B/1	A/1
B	C/0	D/1
C	C/0	D/0
D	B/1	A/0

Máquina M_2

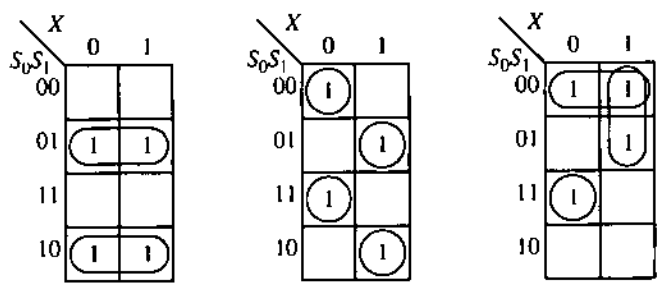
Entrada / S_0S_1	X = 0	X = 1
A: 00	01/1	00/1
B: 01	10/0	11/1
C: 10	10/0	11/0
D: 11	01/1	00/0

Asignación de estados SA_0

Entrada / S_0S_1	X = 0	X = 1
A: 00	01/1	00/1
B: 01	11/0	10/1
C: 11	11/0	10/0
D: 10	01/1	00/0

Asignación de estados SA_1

Figura B.26 • Dos asignaciones de estado diferentes para una misma máquina M_2.

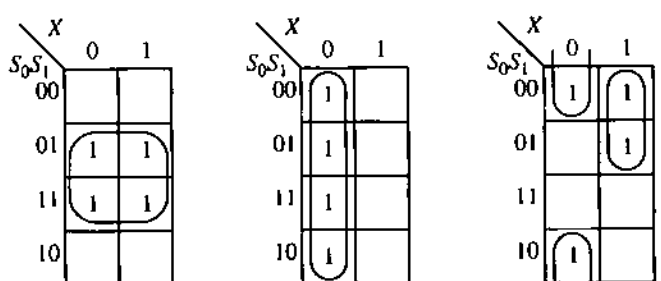


$S_0 = \overline{S}_0S_1 + S_0\overline{S}_1$

$S_1 = \overline{S}_0\overline{S}_1\overline{X} + \overline{S}_0S_1X + S_0S_1\overline{X} + S_0\overline{S}_1X$

$Z = \overline{S}_0\overline{S}_1 + \overline{S}_0X + S_0S_1\overline{X}$

Figura B.27 • Ecuaciones booleanas para la máquina M_2 usando la asignación de estados SA_0.



$S_0 = S_1$

$S_1 = \overline{X}$

$Z = \overline{S}_1\overline{X} + \overline{S}_0X$

Figura B.28 • Ecuaciones booleanas para la máquina M_2 usando la asignación de estados SA_1.

La asignación de estados SA_1 es mucho mejor que SA_0 en términos de la cantidad de entradas del circuito, pero esto no significa que sea mejor si se consideran otros criterios. Por ejemplo, si se realiza la implementación con multiplexores de 8 entradas, la cantidad de entradas de compuertas será la misma para ambas asignaciones. Como consideración adicional, no es simple encontrar la mejor asignación para un criterio determinado. En efecto, en algunos casos pueden lograrse circuitos con una mejor cantidad de entradas si se aumenta la cantidad de bits de estado.

Ejemplo de reducción: un detector de secuencia

En esta sección se establece una vinculación entre los métodos de reducción de las secciones anteriores. La máquina que se desea diseñar entrega un 1 en su salida cuando exactamente dos de sus tres últimas entradas han sido 1 (esta máquina apareció como ejemplo en el apéndice A). Una secuencia de 011011100 produce una secuencia de salida de 001111010. La línea de entrada es única y se puede suponer que no se han visto entradas previas al inicio.

El proceso comienza construyendo un diagrama de transiciones entre estados, como el de la figura B.29. Existen ocho posibles secuencias de tres bits que la máquina debe observar: 000, 001, 010, 011, 100, 101, 110 y 111. El estado *A* es el estado inicial, en el que se supone que aún no se han visto entradas. En los estados *B* y *C* se ha visto una sola entrada, por lo que aún no se puede generar un 1 en la salida. En los estados *D*, *E*, *F* y *G* se han visto solo dos entradas, por lo que no puede generarse aún una salida en 1, a pe-

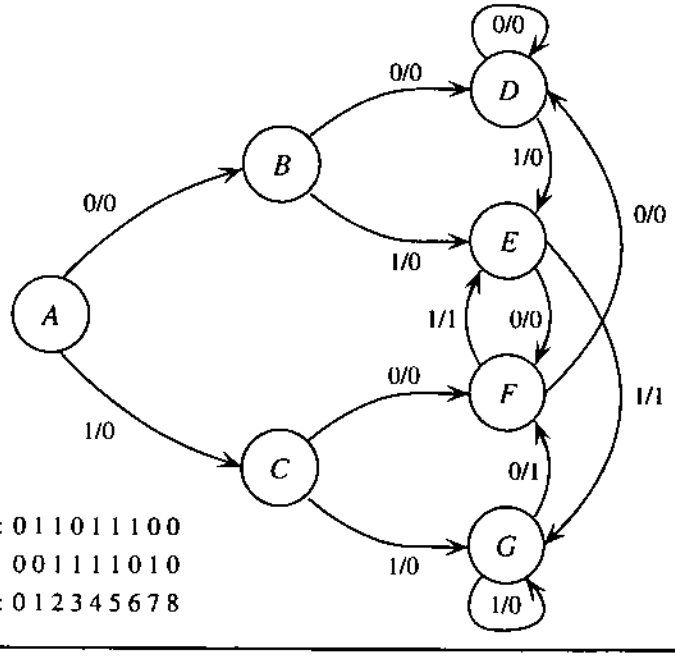


Figura B.29 • Diagrama de transiciones de estados para el detector de secuencia.

sar de haber visto dos unos en la entrada al llegar al estado *G*. La máquina realiza todas las transiciones subsiguientes entre los estados *D*, *E*, *F* y *G*. El estado *D* es el estado al que se llega cuando las dos últimas entradas han sido 0. Los estados *E*, *F* y *G* representan los casos en que las dos últimas entradas han sido 01, 10 o 11, respectivamente.

El próximo paso consiste en la creación de una tabla de estados y la reducción de la cantidad de estados. La tabla de estados de la figura B.30 se toma directamente del diagrama de estados. Se aplica luego la técnica de reducción de estados, para lo que se supone que todos los estados son equivalentes, tras lo cual se refina la hipótesis. El proceso es el siguiente:

$$P_0 = (ABCDEFG)$$
$$P_1 = (ABCD)(EF)(G)$$
$$P_2 = (A)(BD)(C)(E)(F)(G)$$
$$P_3 = (A)(BD)(C)(E)(F)(G)\ \surd$$

Estado actual \ Entrada	X = 0	X = 1
A	*B*/0	*C*/0
B	*D*/0	*E*/0
C	*F*/0	*G*/0
D	*D*/0	*E*/0
E	*F*/0	*G*/1
F	*D*/0	*E*/1
G	*F*/1	*G*/0

Figura B.30 • Tabla de estados para el detector de secuencia.

En el trayecto 0,0,0 del diagrama de transiciones, los estados *B* y *D* son equivalentes. Se genera una nueva tabla de transiciones para indicar los nuevos estados, la que se muestra en la figura B.31.

Estado actual \ Entrada	X = 0	X = 1
A: *A'*	*B'*/0	*C'*/0
BD: *B'*	*B'*/0	*D'*/0
C: *C'*	*E'*/0	*F'*/0
E: *D'*	*E'*/0	*F'*/1
F: *E'*	*B'*/0	*D'*/1
G: *F'*	*E'*/1	*F'*/0

Figura B.31 • Tabla de estados reducida del detector de secuencia.

Luego se realiza una asignación de estados arbitraria, como la de la figura B.32. Esa asignación de estados se utiliza para generar los mapas K de las funciones correspondientes a los estados futuros y a la salida, los que se muestran en la figura B.33. Nótese la exis-

tencia de cuatro estados indeterminados por la falta de uso de las asignaciones correspondientes a 110 y 111. Por último, se implementa el circuito a nivel de compuertas, el que se ilustra en la figura B.34. •

Estado actual \ Entrada	X = 0	X = 1
$S_2S_1S_0$	$S_2S_1S_0Z$	$S_2S_1S_0Z$
A': 000	001/0	010/0
B': 001	001/0	011/0
C': 010	100/0	101/0
D': 011	100/0	101/1
E': 100	001/0	011/1
F': 101	100/1	101/0

Figura B.32 • Asignación de estados del detector de secuencia.

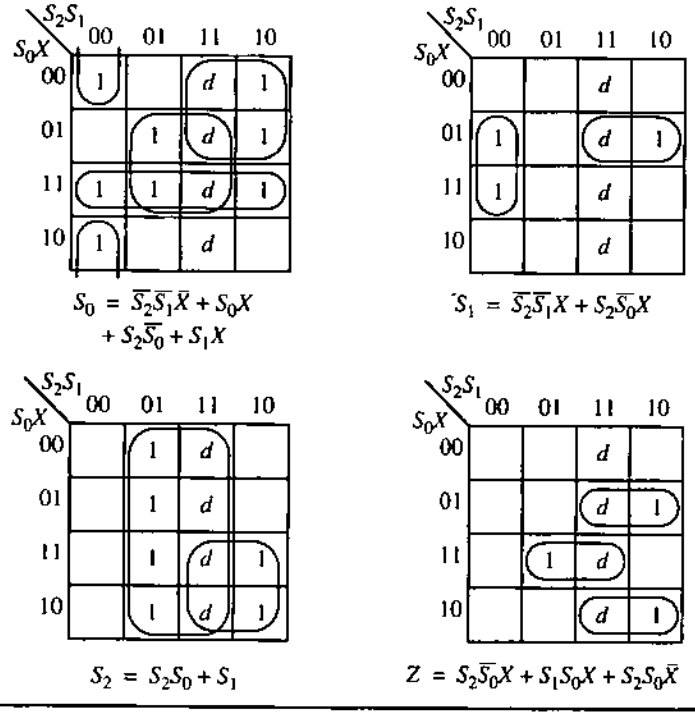


Figura B.33 • Simplificación de las funciones de salida y estados futuros por medio de mapas K.

B.3.2 Tablas de transiciones

Además del uso de los circuitos S-R y D, son de uso común los *flip flops* **J-K** y **T** (véase el apéndice A). El *flip flop* J-K se comporta en forma similar al *flip flop* S-R, excepto porque utiliza el estado 11 de las entradas para conmutar su estado. El *flip flop* T (*toggle*) al-

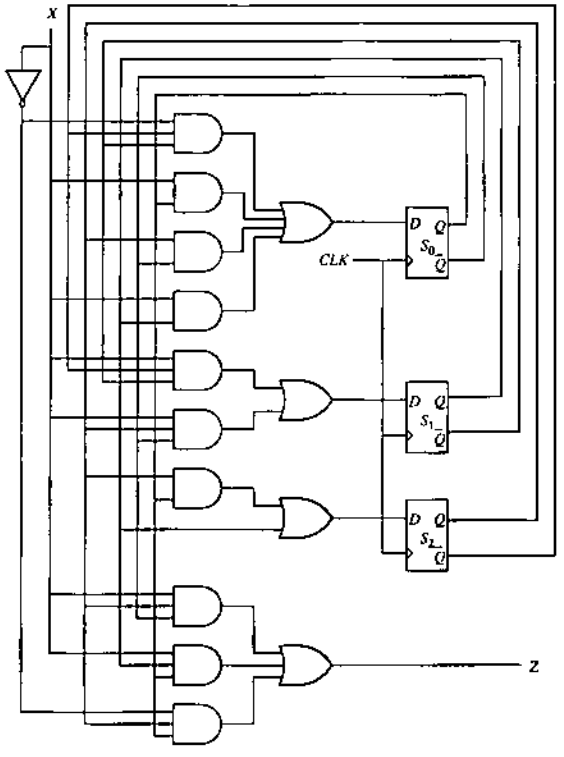


Figura B.34 • Implementación del detector de secuencia a nivel de compuertas.

terna sus estados, tal como ocurre con el *flip flop* J-K cuando se conectan sus dos entradas a 1. Los diagramas lógicos y símbolos de los dos circuitos se ilustran en las figuras B.35 y B.36, respectivamente.

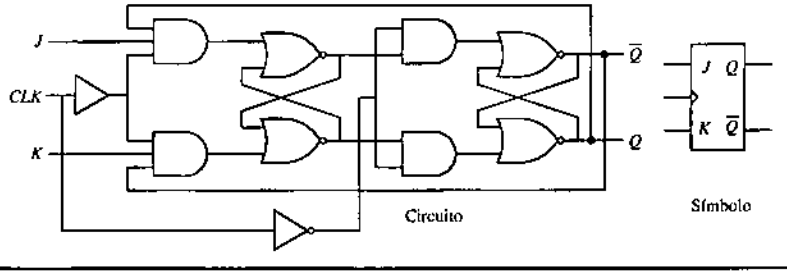


Figura B.35 • Circuito lógico y diagrama de un *flip flop* J-K.

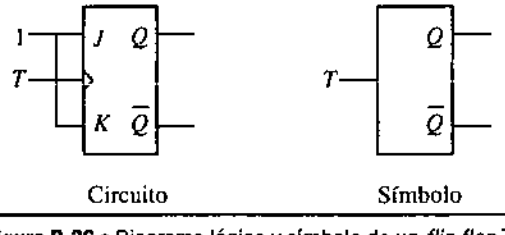


Figura B.36 • Diagrama lógico y símbolo de un *flip flop* T.

Todos los *flip flops* analizados hasta este momento, así como algunas variantes, se consiguen como componentes separados, por lo que, en épocas pasadas, los diseñadores optaban por una forma u otra en función de características de costo, eficiencia, disponibilidad, etc. En la actualidad, con el desarrollo de circuitos integrados VLSI, se suele usar el *flip flop* D en gran cantidad de circuitos. Los circuitos integrados de alta velocidad que utilizan lógica de baja densidad, como los de arseniuro de galio (GaAs), todavía pueden encontrar aplicaciones para las diferentes formas. Para situaciones como esta, se considera el problema de encontrar un *flip flop* que minimice el número total de componentes de un circuito, considerando al *flip flop* como componente individual.

Los cuatro *flip flops* analizados hasta aquí pueden describirse por medio de **tablas de transiciones**, como las que se indican en la figura B.37. Cada tabla describe las condiciones que deben aplicarse en las entradas en el instante t para cambiar la salida en el instante $t+1$.

flip-flop S-R

Q_t	Q_{t+1}	S	R
0	0	0	0
0	1	1	0
1	0	0	1
1	1	0	0

flip-flop D

Q_t	Q_{t+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

flip-flop J-K

Q_t	Q_{t+1}	J	K
0	0	0	d
0	1	1	d
1	0	d	1
1	1	d	0

flip-flop T

Q_t	Q_{t+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Figura B.37 • Tablas de transiciones para los cuatro tipos de *flip flop*.

Como ejemplo del uso de las tablas de transiciones en el diseño de una máquina de estados finitos, considérese el uso de un *flip flop* J-K en el diseño de un sumador serie. Se inicia el proceso con el sumador serie de la figura B.38. El estado *A* representa el caso en el que no existe arrastre generado en el instante de tiempo anterior, en tanto que el estado *B* representa el caso en que sí existe arrastre desde el instante anterior.

Luego se crea la tabla de verdad para el *flip flop* correspondiente. La figura B.39 muestra la tabla de verdad que especifica las funciones de los *flip flops* D, S-R, T y J-K, así como su salida Z. En este caso, solo se analizarán las funciones correspondientes al *flip flop* J-K y a la salida Z.

La forma en que se construye la tabla de estados parte de observar primero el valor del estado actual S_i y compararlo con el valor deseado para el estado siguiente. Luego se utilizan las tablas de transiciones para determinar en forma coherente los valores de entradas de los *flip flops*. Por ejemplo, en la primera línea de la tabla de verdad de la figura

Estado actual \ Entrada	XY = 00	01	10	11
A	A/0	A/1	A/1	B/0
B	A/1	B/0	B/0	B/1

Próximo estado Salida

Estado actual (S_t) \ Entrada	XY = 00	01	10	11
A:0	0/0	0/1	0/1	1/0
B:1	0/1	1/0	1/0	1/1

Figura B.38 • Diagrama de estados, tabla de estados y asignación de estados para un sumador serie.

B.39, cuando $X = Y = 0$ y el estado actual es 0, el estado siguiente debe ser 0 tal como se verifica en la tabla de estados de la figura B.38. Con el objeto de lograr esta transición en un *flip flop* J-K, las entradas J y K deben ser, respectivamente, 0 y d, tal como se lee de la tabla de transiciones del *flip flop* J-K de la figura B.39. Si se procede en forma similar, se completa la tabla de verdad, donde se obtienen las ecuaciones booleanas reducidas que se indican a continuación:

$$J = XY$$

$$K = \bar{X}\bar{Y}$$

$$Z = \bar{X}\bar{Y}S + \bar{X}Y\bar{S} + XYS + X\bar{Y}\bar{S}$$

Estado actual X	Y	S_t	(Set) D	S	(Reset) R	T	J	K	Z
0	0	0	0	0	0	0	0	*d*	0
0	0	1	0	0	1	1	*d*	1	1
0	1	0	0	0	0	0	0	*d*	1
0	1	1	1	0	0	0	*d*	0	0
1	0	0	0	0	0	0	0	*d*	1
1	0	1	1	0	0	0	*d*	0	0
1	1	0	1	1	0	1	1	*d*	0
1	1	1	1	0	0	0	*d*	0	1

Figura B.39 • Tabla de verdad que ilustra los estados futuros de un sumador serie usando *flip flops* D, S-R, T y J-K. Las funciones sombreadas son las que se usan en el ejemplo.

El circuito correspondiente se muestra en la figura B.40. Nótese que el diseño tiene una cantidad relativamente baja de entradas de compuertas (20), si se lo compara con el cir-

cuito lógico equivalente de la figura B.41, el que utiliza un *flip flop* D. Los *flip flops* no se incluyen en la cuenta del número de entradas, aunque sí contribuyen a la complejidad circuital.

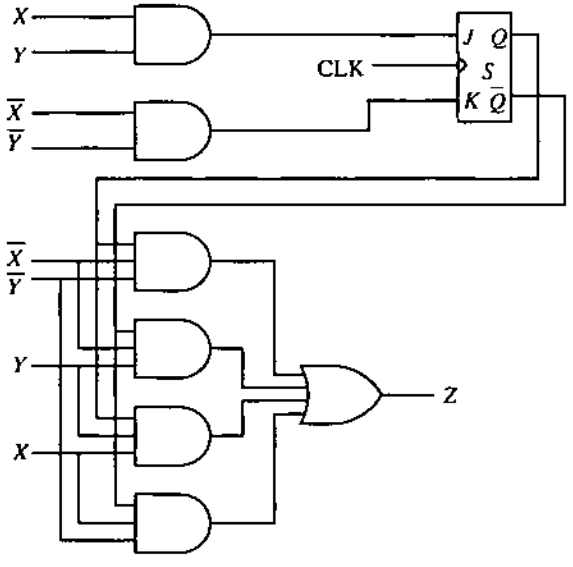


Figura B.40 • Sumador serie que utiliza un *flip flop* J-K.

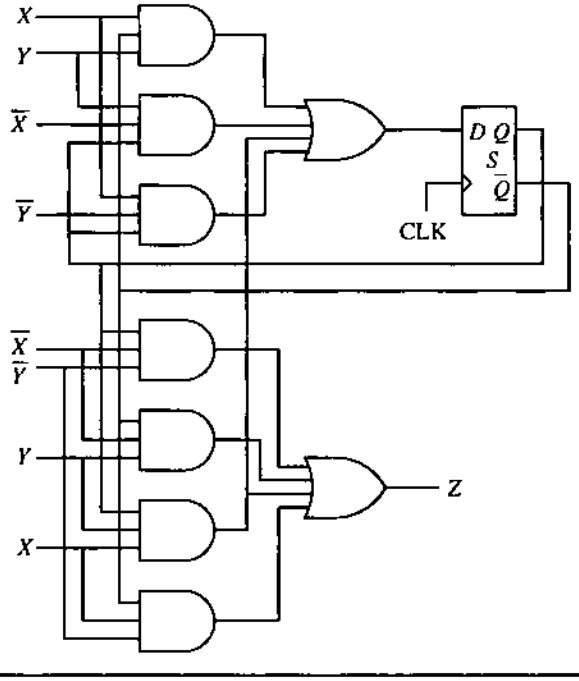


Figura B.41 • Sumador serie que utiliza un *flip flop* D.

Ejemplo de tabla de transiciones: un detector de mayoría

Para este ejemplo, se encarará el diseño de un circuito que utiliza *flip flops* T y multiplexores de ocho entradas de datos para la generación de la función mayoría (véase la figura A.15) para tres entradas que se ingresan en serie a una máquina de estados finitos. El circuito entrega un 0 en su salida hasta el momento de recibir el tercer bit de entrada, momento en el cual entregará un 0 o un 1 en su salida si hubo más ceros o más unos en la entrada, respectivamente. Por ejemplo, una entrada de 011100101 produce una salida de 001000001.

El método se inicia con la creación de un diagrama de transiciones de estados que enumera todos los estados posibles de la máquina de estados finitos. En la figura B.42 se muestra un diagrama de estados en el que la organización de dichos estados depende de la cantidad de entradas observadas. El estado *A* es el estado inicial, en el que no se ha recibido entrada alguna. Las tres entradas se simbolizan con la notación xxx. Luego de recibir la primera entrada, la máquina de estados finitos realiza una transición al estado *B* o al estado *C*, según que la entrada sea 0 o 1, respectivamente. El historial de entradas se representa con las notaciones 0xx y 1xx, respectivamente, para los estados *B* y *C*. Los estados *D*, *E*, *F* y *G* enumeran todos los posibles historiales de dos entradas, lo que se simboliza con las notaciones 00x, 01x, 10x y 11x, respectivamente.

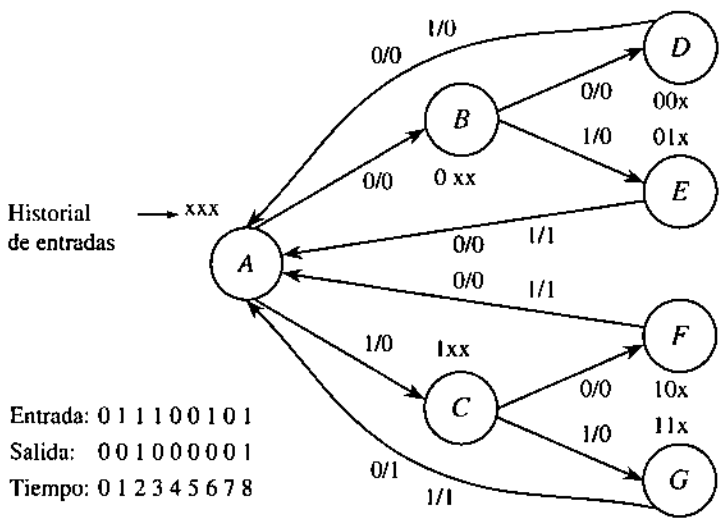


Figura B.42 • Diagrama de estados de la función mayoría implementada por una máquina de estados finitos.

La salida de la máquina de estados finitos es 0 cuando se realizan transiciones hacia los estados *B* a *G*. En la tercera entrada, la máquina vuelve al estado *A*, entregando un 0 o un 1 en su salida de acuerdo con la función mayoría. La máquina de estados finitos de la figura B.42 utiliza un total de siete estados, los que se resumen en la tabla de estados de la figura B.43a.

Estado actual \ Entrada	X = 0	X = 1
A	B/0	C/0
B	D/0	E/0
C	F/0	G/0
D	A/0	A/0
E	A/0	A/1
F	A/0	A/1
G	A/1	A/1

(a)

$P_0 = (ABCDEFG)$
$P_1 = (ABCD)(EF)(G)$
$P_2 = (AD)(B)(C)(EF)(G)$
$P_3 = (A)(B)(C)(D)(EF)(G)$
$P_4 = (A)(B)(C)(D)(EF)(G)$ √

(b)

Estado actual \ Entrada	X = 0	X = 1
A: A'	B'/0	C'/0
B: B'	D'/0	E'/0
C: C'	E'/0	F'/0
D: D'	A'/0	A'/0
EF: E'	A'/0	A'/1
G: F'	A'/1	A'/1

(c)

Figura B.43 • (a) Tabla de estados; (b) particiones; (c) tabla de estados reducida.

La máquina de siete estados puede reducirse a un esquema de seis estados. El proceso de reducción se ilustra en la figura B.43b. Los estados *E* y *F* pueden combinarse en uno solo, de acuerdo con lo que ilustra la tabla reducida de la figura B.43c. La tabla reducida de estados se utiliza para crear la asignación de estados, la que se ilustra en la figura B.44a considerando el uso de *flip flops* D. Dado que se desea utilizar *flip flops* T, mediante la misma asignación de estados se obtiene la tabla de estados de la figura B.44b. La versión de la implementación con *flip flops* T se obtiene comparando el estado actual con el estado futuro y siguiendo la tabla de transiciones del *flip flop* T, que se muestra en la figura B.37. Si se toma como base la versión del circuito implementada con *flip flops* D, se podrá ver la diferencia de comportamiento del *flip flop* T: si los estados actual y siguiente de la implementación D coinciden, el estado siguiente en la versión implementada con *flip flops* T será 0; si los estados actual y siguiente de la versión implementada con *flip flops* D difieren, la versión T adoptará un estado 1. Se utilizan tres bits para la codificación binaria de cada estado, en consecuencia, se tienen tres funciones para definir los estados siguientes (s_0, s_1 y s_2) y una función de salida Z. El circuito correspondiente que utiliza *flip flops* T se muestra en la figura B.45. A los estados irrelevantes 110 y 111 se les ha asignado el valor cero. •

Estado actual \ Entrada	X = 0	X = 1
$S_2S_1S_0$	$S_2S_1S_0Z$	$S_2S_1S_0Z$
A': 000	001/0	010/0
B': 001	011/0	100/0
C': 010	100/0	101/0
D': 011	000/0	000/0
E': 100	000/0	000/1
F': 101	000/1	000/1

(a)

Estado actual \ Entrada	X = 0	X = 1
$S_2S_1S_0$	$T_2T_1T_0Z$	$T_2T_1T_0Z$
A': 000	001/0	010/0
B': 001	000/0	010/0
C': 010	110/0	111/0
D': 011	011/0	011/0
E': 100	100/0	100/1
F': 101	101/1	101/1

(b)

Figura B.44 • Asignación de estados de la máquina de estados finitos reducida que implementa la función mayoría, (a) con *flip flops* D y (b) con *flip flops* T.

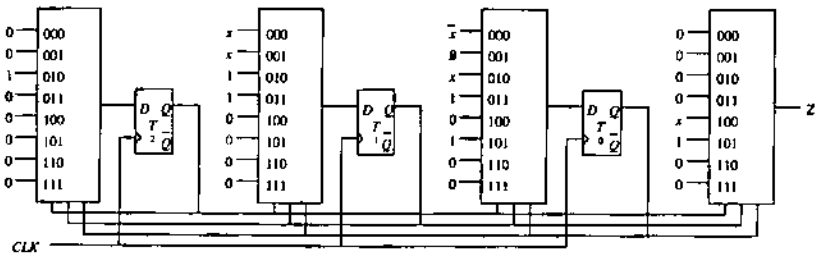


Figura B.45 • Circuito lógico de la máquina de estados que implementa la función mayoría.

Resumen

Los circuitos que se obtienen a partir de ecuaciones no simplificadas pueden ser muy grandes, por lo que, en lo posible, las expresiones lógicas deben reducirse para convertirlas en ecuaciones lógicamente equivalentes pero más pequeñas. Una forma de reducir expresiones utiliza las propiedades del álgebra de Bool para realizar operaciones algebraicas. Este método es muy poderoso, pero requiere mecanismos de prueba y error, resultando tedioso para su realización manual. Un método más simple es el que utiliza para la simplificación los denominados mapas K. Esta técnica, más visual, se vuelve complicada cuando se pretende utilizarla para más de seis variables. El método tabular permite su automatización y ofrece la posibilidad de compartir términos entre funciones.

Una máquina de estados finitos solo puede adoptar, en un momento determinado, uno de un determinado conjunto de estados, pero existen muchas máquinas de estados finitos que, miradas desde el exterior, manifiestan el mismo comportamiento. La cantidad de *flip flops* requeridos por una máquina de estados finitos puede reducirse a través del proceso de simplificación de sus estados, y la complejidad de la lógica combinatoria de la máquina puede reducirse por medio de una asignación de estados adecuada. También la elección del tipo de *flip flop* a utilizar condiciona la complejidad del circuito resultante. Se suele usar el *flip flop* D como elemento para la implementación de las máquinas de estados finitos, aun cuando podrían utilizarse otros *flip flops* como el S-R, el J-K o el T.

Para lectura posterior

T. L. Booth ofrece una buena explicación del procedimiento de simplificación de Quine-McCluskey. Z. Kohavi ofrece un profundo tratamiento de los métodos de simplificación tanto en lógica combinatoria como de estados. El texto de V. C. Hamacher y otros contiene un análisis interesante de los temas referidos a temporización y retardos de propagación. C. Mead y L. Conway describen en forma concisa las técnicas de diseño VLSI. V. D. Agrawal y K. T. Cheng cubren el diseño basado en asignación de estados.

Agrawal, V. D. y K. T. Cheng, "Finite state machine synthesis with embedded test function," en: *Journal of Electronic Testing: Theory and Applications*, vol. 1, 1990, p.p. 221-228.

Booth, T. L., *Introduction to Computer Engineering: Hardware and Software Design*, 3a. ed., John Wiley & Sons, 1984.

Hamacher V. C, Z. G. Vranesic y S. G. Zaky, *Computer Organization*, 3a. ed., McGraw Hill, 1990.

Mead, C. y L. Conway, *Introduction to VLSI Systems*, Addison Wesley, 1980.

Kohavi, Z., *Switching and Finite Automata Theory*, 2a. ed., McGraw Hill, 1978.

Problemas

B.1 Dadas las siguientes funciones, construir los mapas K y encontrar las funciones mínimas representativas de f y g.
$f(A,B,C,D) = 1$ si dos o más entradas son 1, cero en caso contrario.
$g(A,B,C,D) = 1$ cuando la cantidad de entradas en 1 es par (incluyendo el caso en que no haya ningún 1 en la entrada), en caso contrario $g(A,B,C,D) = \overline{f(A,B,C,D)}$.

B.2 Mediante mapas K, simplificar la función f siguiente considerando los estados irrelevantes indicados. Realizar la simplificación tanto en la forma suma de productos como en la forma producto de sumas.

$$f(A,B,C,D) = \sum(2,8,10,11) + \sum_{d}(0,9)$$

B.3 Dado un circuito lógico, ¿es posible generar una tabla de verdad que lo represente y que presente estados irrelevantes? Justificar la respuesta.

B.4 El mapa K de la figura está mal confeccionado. Indicar la ecuación reducida que se obtiene de esta configuración errónea. Generar el mapa K correcto y deducir, del mapa correcto, la ecuación simplificada. Nótese que ambos mapas K generarán ecuaciones funcionalmente correctas, pero solo la función obtenida desde el mapa correcto entregará la función minimizada.

D \ ABC	000	001	011	010	110	111	101	100
0	1			1	1			1
1	1			1	1			1

B.5 La tabla de verdad siguiente representa un multiplexor de cuatro entradas y una salida. Utilizando variables ingresadas en el mapa, generar una ecuación booleana en la forma suma de productos.

A	B	F
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

B.6 Utilizar el método tabular para simplificar la función siguiente

$$f(A,B,C,D) = \sum_m (3,5,7,19,13,15) + \sum_d (2,6)$$

B.7 Utilizar el método tabular para reducir las funciones representadas por la siguiente tabla de verdad de tres salidas:

Término mínimo	A	B	C	D	F_0	F_1	F_2
m_0	0	0	0	0	0	0	1
m_1	0	0	0	1	0	0	0
m_2	0	0	1	0	0	0	0
m_3	0	0	1	1	1	0	0
m_4	0	1	0	0	0	0	1
m_5	0	1	0	1	1	1	0
m_6	0	1	1	0	0	0	0
m_7	0	1	1	1	1	1	0
m_8	1	0	0	0	0	0	0
m_9	1	0	0	1	0	0	0
m_{10}	1	0	1	0	0	1	1
m_{11}	1	0	1	1	0	0	0
m_{12}	1	1	0	0	0	0	0
m_{13}	1	1	0	1	1	1	0
m_{14}	1	1	1	0	1	1	1
m_{15}	1	1	1	1	1	1	1

B.8 Reducir la siguiente ecuación de F a su forma mínima en dos niveles, e implementarla con una matriz lógica programable de tres entradas y una salida.

$$F(A, B, C) = ABC + \bar{A}BC + A\bar{B}\bar{C} + \bar{A}\bar{B}C$$

B.9 Utilizar el método de descomposición de funciones para implementar la función f indicada con dos multiplexores de cuatro entradas de datos. Organizar la ecuación de modo tal que C y D queden en el nivel interior del circuito, tal como ocurre en la ecuación B.22. Cada entrada de cada uno de los multiplexores debe quedar asignada a un valor lógico (0 o 1), a una variable o a una función.

$$f(A, B, C, D) = ABCD + AB\bar{C}D + ABC\bar{D} + \bar{A}B$$

B.10 Reducir la tabla de estados siguiente:

Entrada / Estado actual	X = 0	X = 1
A	D/0	G/1
B	C/0	G/1
C	A/0	D/1
D	B/0	C/1
E	A/1	E/0
F	C/1	F/0
G	E/1	G/1

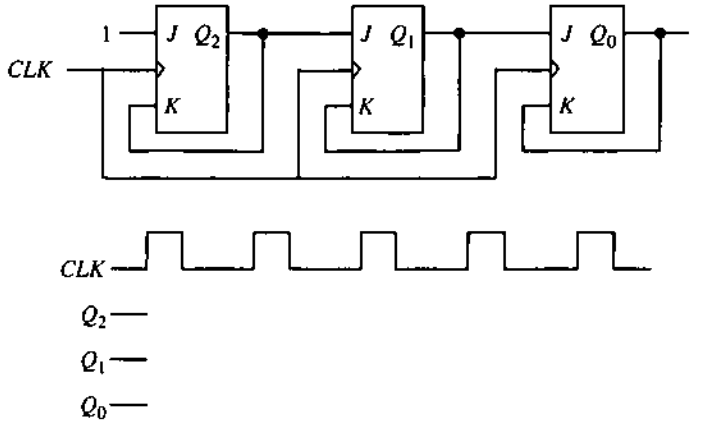
B.11 Reducir la tabla de estados siguiente:

Entrada / Estado actual	XY = 00	01	10	11
A	A/0	B/0	C/0	D/0
B	A/0	B/1	D/0	D/1
C	E/1	B/0	B/0	E/1
D	A/0	D/1	D/0	B/1
E	C/1	D/0	D/0	E/1

B.12 La tabla de estados siguiente puede o no ser simplificada. Indicar el proceso de simplificación (generar las particiones) y la tabla de estados reducida.

Entrada / Estado actual	x = 0	1	2
A	B/0	E/2	G/1
B	D/2	A/1	D/0
C	D/2	G/1	B/0
D	B/2	F/1	C/0
E	A/0	E/2	C/1
F	C/0	E/2	F/1
G	D/0	E/2	A/1

B.13 El circuito siguiente contiene tres *flip flops* J-K maestro-esclavo. Hay una única entrada *CLK* y tres salidas Q_2, Q_1 y Q_0 las que inicialmente adoptan el valor 0. Completar el diagrama de tiempos indicando los valores de Q_2, Q_1 y Q_0, suponiendo que no existan retardos entre los *flip flops*.



B.14 Utilizar un *flip flop* T para implementar un sumador serie, siguiendo el esquema de la sección B.3.2.

B.15 En la tabla de estados reducida que se ilustra, se han realizado las asignaciones de estado correspondientes. Diseñar la máquina usando *flip flops* D y compuertas Y y O. Utilizar mapas K para reducir las expresiones correspondientes a la salida y a los estados siguientes. Para construir los mapas K debe tenerse en cuenta que solo hay tres renglones en la tabla de estados.

Entrada / Estado actual	X = 0	X = 1
YZ		
A: 00	00/0	01/1
B: 01	10/1	00/1
C: 10	01/1	10/0

B.16 Dibujar un diagrama lógico que permita implementar un *flip flop* J-K utilizando un *flip flop* D.

Índice analítico

360, IBM, 145
68000, Motorola, 145, 436
80486, Intel, 107
8080, Intel, 145
80x86, Intel, 107, 145

A

acceso
  directo a memoria por robo de ciclos, 313
  múltiple por detección de portadora, 358
acierto
  memoria *cache*, 257
  tasa de, 265
activo
  alto, 456
  bajo, 457
acumulador, 127
Adobe PostScript, 339
AGP, 307
AHPL, 219
aleatorio
  memoria de acceso, 245
  política de reemplazo, 259
algoritmo
  de Booth, 80
  de encaminamiento fuera de línea, 419
  de verificación, 327
  Neat Little de reemplazo, 269
alineación
  forzada, 315
  no forzada, 315
alta impedancia, 449
AltiVec, 177-185
ALU, 253
  como parte de la computadora, 4
AM, modulación en amplitud, 348
análisis
  lexicográfico, en compilación, 152
  sintáctico, en compilación, 152
analizador, 152
ancho de banda, 341, 371
anillo
  en el procesamiento paralelo, 415
  topología dc redes, 357
Apple Computer,
  emulación del 68000, 11
Apple Macintosh, 107
arbitraje
  bus de, 305
  centralizado, 305
  descentralizado, 305
árbol
  decodificador, 249, 279
  desbalanceado, 520
  grueso, 432
ARC, 108-121
  conjunto de instrucciones, 110-112
  descripción de instrucciones, 117-121
  entrada-salida mapeada en memoria, 109
  formatos de datos, 115-117
  formatos de instrucciones, 113-115
  instrucciones
    `addcc`, 119
    `andcc`, 118
    `ba`, 121
    `bcs`, 120
    `be`, 120
    `bne`, 158
    `bneg`, 120
    `bvs`, 120
    `call`, 119

jmpl, 119
ld, 117
orcc, 118
orncc, 118
rett, 218, 312
sethi, 118
smul, 150
srl, 119
st, 118
subcc, 150
xorcc, 239
lenguaje simbólico, 112-113
memoria, 109
modos de direccionamiento, 124
archivo, 325
área de circuito liberada, 390
aritmética de saturación, 182
arquitectura
de procesador escalable, 108
de programación, 11
de programación, 99
unidad de, IU, 406
Harvard, 265
normalizada industrial, ISA, 309
arrastre circular, 67
arreglo sistólico, 418
ASCII, 50
ASIC, 228
asignación
asociativa por conjuntos, 262
de acciones, 152
asincrónico
bus, 303
entrada-salida, 299
modo de transferencia, ATM, 378
asociativa
asignación de *cache*, 256
memoria, 258, 281
teorema, 451
aumento de velocidad, 390, 415

B

B5000, 145
Babbage, Charles, 2, 145
base
conversión entre, *véase* número, conversión
definición, 22
BER, tasa de error de bits, 360
biestable *set-reset*, 472
big endian, 102, 109
binario,
código, 9
compatibilidad, 9
decimal codificado, 34
números, 20
BISDN, 379
bit
codificación por pares, 80
de paridad, 361
de suciedad, 257
de validez, 257
implícito, en números de punto flotante, 37
tableta digitalizadora, 335
tasa de error, 360
bloque
error, 368
maestro de control, 326
secuencia de control, 368
tasa, velocidad de transferencia de disco, 323
bloqueo en una red, 418
bloques
en memoria asociativa, 256
memoria, 249
brazo, disco magnético, 321
BRI, 379
buffer circular, 400
burbuja
coincidencia, 457
Burroughs B5000, 145
bus, 300
arbitraje, 217, 305
árbitro, 305
ciclo, 302
compartido, 256
concesión, 305
de control, 5
de datos, 5
de direcciones, 5
de entrada de datos, 5
de salida de datos, 5
definición, 5
del sistema, 5, 100
modelo, 5
frontal, 307
línea de pedido, 305
maestro, 301
posterior, 307
serie universal, 309
sistema, 5
topología, 356
byte, 101
bytecode, 139

C

cabeza
  disco magnético, 321
  móvil, 320
cache, 254-269
  asignación asociativa, 256
  asignación y escritura, 265
  bloques, 256
  escritura diferida, 265
  escritura inmediata, 265
  escritura sin asignación, 265
  fraccionada, 265
  líneas, 256
  memoria, 254
  multinivel, 267
  rendimiento, 263
  tasa de aciertos, 265
  tiempo de acceso, 265
caja negra, 461
calculadora HP 9100A, 89
cálculo de residuos, 85
CAM, 260, 281
campo de etiquetas, 257
campos, en memorias asociativas, 282
canal, 371
canónica
  forma de una ecuación booleana, 455
  forma producto de sumas, 455
  forma suma de productos, 453
capacidad de carga (*fan out*), 460
capacitor, 246
carga, 151
  en paquetes de datos, 354
  especulativa, 413
  identificador de tipo, en paquetes ATM, 381
  inmediata, 264
  y almacenamiento, 110
cargador de reubicación, 172
cargador, 168, 171
CD, 330
CD-ROM, 330
ciclo de búsqueda-ejecución, 105
cilindro, disco magnético, 322
circuito
  de niveles múltiples 442,
  integrado de muy alta velocidad, VHSIC, 219
  integrado, 12, 301, 288
  lógico, 10,
circuitos
  analógicos, 442
  integrados en gran escala, 283, 461
CISC
  arquitectura Intel Pentium como CISC, 179
  y RISC, 391-393
CLP, en paquetes ATM, 381
codificación *one hot*, 221
codificador, 465
  de prioridad, 465
código de corrección de errores Reed Solomon, 331
código Gray, 502
códigos de condición, 110
Colossus, computadora, 3
Compatibilidad
  binaria, 9
  de código fuente, 8
  hacia arriba, 7
compilación, 151-160
  cruzada, 153
  en silicio, 219
compilador, 107
  definición, 8
compilador justo a tiempo, 141
complemento
  a dos, 32
  a uno, 32, 444, 451
componentes digitales, 460
compuerta
  cantidad de entradas, 456
  cantidad, 456
  lógica, 445
  retardo, 508
computadora de programa almacenado, 5
computadora TYAN, 13
comunicaciones celulares, 353
concentrador, 357
conexiones, 373
configuración de línea encadenada, 305
conjunto
  de trabajo, 273
  elegible, 513
constante
  conjunto de constantes, en la máquina virtual Java, 140
  velocidad angular, disco, 331
  velocidad lineal, disco, 331

contador
de posición, 112, 162
de programa, 104, 112
control
bus de, 5
de errores de encabezado, HEC, en paquetes ATM, 381
de redundancia cíclica, 368, 369
genérico de flujo, en redes ATM, 381
lógico de enlace, 356
memoria, 201
sección, 104, 192
señales, 9
unidad de
cableado, 192
como parte de la computadora, 4
definición, 9
microprogramado, 192
controlador
de interrupciones de periféricos, *pic*, 312
en manejo de superposiciones, 270
controladores de dispositivos, 300
convención de llamada, 130
corrección de errores, SEC, 365
correo electrónico, 355
CPU, 104
como parte de la computadora, 4
CRC, 368, 369
crossbar, 418
CSMA/CD, 358

D

datos
bus, 5
codificación, 19
hoja de, 458
representación, 19
sección de (CPU), 104, 192
zona de transferencia, 130
decodificador, 464
demultiplexor, 463
dependencia
análisis, 424
entre instrucciones, 406
gráfico, 425
desborde, 20, 64, 117
desensamblador, 187
detección de colisiones, 358
diagrama de Venn, 501
diámetro de una red, 418
digital
circuito, 441
disco versátil, DVD, 330, 332
dígito de guarda, 89
dinámica
biblioteca de enlace, 168
memoria RAM, 246
diodo emisor de luz, 335
dirección, 102
bus, 5
de entrada-salida, 6
espacio, 102, 103
direccionamiento base, 154
directo
acceso a memoria, 309, 312-313
modo de direccionamiento, 129
disco
compacto, 330
floppy, 325
discos magnéticos, 320
grabación por zonas, 322
disquete, *floppy*, 325
división con reposición, 72
DLL, 168
DMA, 309, 312, 313
robo de ciclos, 313
DRAM, 246
DVD, 330, 332

E

EBCDIC, 52
Eckert, J. Presper, 3
ECMA-23, 334
editor de enlace, 168
EDSAC, 4
EDVAC, 4, 144
EEPROM, 253
eficiencia, 415, 416
eficiencia de ejecución, 397
eje, 320
ejecución
concurrente, VHDL, 228
superescalar, 387
elemento de imagen, 340
elementos de procesamiento, 415
emulación, 11, 216
emulador, 11
en números de punto flotante, 35
encaminadores, 356, 375
encapsulado, 374

encriptado, primeros intentos, 3
encuesta, 309
ENIAC, 3
ENIGMA, 3
enlace, 151
  cargador, 171
enlace entre subrutinas, 130
ensamblado
  lenguaje,
    definición, 9
    vs. lenguaje de máquina, 99
  luego de la compilación, 108
  proceso, 151, 160
ensamblador, 9
entrada-salida programada, 309
entrelazado, 249, 326
EPIC, procesamiento de instrucciones explícitamente paralelo, 411
EPROM, 253
error
  en números de punto fijo, 21
  en números de punto flotante, 41
espacio
  entre pistas, 322
  entre sectores, 322
especificaciones de asignación, en compiladores, 153
espera por dispositivo ocupado, 309
estancamiento, 358
estrictamente no bloqueante, 418
estructuras lógicas completas, 452
extensión de signos, en ARC, 115

F

factor de carga equivalente (*fan in*), 460
FCS, 368
fibra
  de modo único, 352
  multimodo, 352
FIFO, 259
fijo
  disco de cabezas, 329
  números de punto, 20-35
    error, 21
    precisión, 21
    rango, 21
finitos
  análisis de elementos, 431
  máquina de estados, 471
firmware, 205
  definición, 10
flecha de causa y efecto, 304
flip flop, 472
flujo múltiple de instrucciones
  flujo múltiple de datos, 417
  flujo único de datos, 418
flujo único de instrucciones
  flujo múltiple de datos, 416
  flujo único de datos, 416
FM, 348
formato Manchester, 321
FPGA, 491
fragmentación,
  disco, 326
  memoria, 276
frecuencia de reloj, 475
función, 130
  lógica booleana, 444
  mayoría, 229, 453
fusión, en ISDN, 379

G

generación
  de código, 152
  en arrastre anticipado, 77
grabación por zonas, en discos magnéticos, 322
gráfico de partición, 270
granularidad, en arquitecturas paralelas, 429
grupo,
  generación, 86
  propagación, 86
  sumador con arrastre anticipado, 86

H

HDL, 219
historia de las computadoras, 1
hoja de datos, 458
huecos, en discos compactos, 330

I

IA 64, 411-414
IBM
  360, 7, 8, 145
  701, 144
  operaciones vectoriales en IBM 370, 178
  texto acerca de, 17
idempotencia, 451
identidad, 450
impedancia, 449
implicante primo esencial, 513
implicantes primos, 502
instrucciones
  conjunto de, 9, 106
  de dos direcciones, 125, 127
  de salto condicional, en ARC, 112
  de tres direcciones, 125, 126
  de una dirección, 125, 127
  dependencia, 406
  derivadas, 406
  formato, ARC, 113
  registro de, 104
integración
  en gran escala, 461
  en media escala, 461
  en pequeña escala, 461
Intel
  8080, 145
  familia de procesadores, 9
  IA-64, Merced, 411-414
  MMX, 177-185
  Pentium II, 14
  Pentium, 178
interconexión
  de componentes periféricos, 308
  de sistemas abiertos, 354
interfaz
  de velocidad básica, 379
  de velocidad primaria, 377
  entre redes, en ATM, 381
  para sistemas pequeños de computación, 308, 324
  usuario-red, 381
International Standard Organization, 354
interrupción no enmascarable, NMI, 312
interrupciones, 216-218
  entrada-salida manejada por, 309
  rutina de atención de, ISR, 311
  vector de, 217
intersecciones
  complejidad, 418
  conmutadores, 418
intervalo entre registros, cinta, 328
inversor, 445
involución, 451
IPv4, 373
IPv6, 373
ISA, arquitectura normalizada industrial, 309
ISDN de banda ancha, 379
ISDN de banda angosta, 379
ISDN, 379
ISO, 354
iteración de Newton, 83

J

Java
  archivo de clase, 139
  máquina virtual, 139-144

L

LAN, 354
lápiz óptico, 336
latencia de rotación, 323
LED, 335
lenguaje
  de descripción de hardware, 218
  microensamblador, 206
ley de Amdahl, 415
LFU, 259
limpieza del *pipeline*, 395
línea de transmisión, 287
líneas, 256
little endian, 102
LLC, 356
localidad
  espacial, 255
  temporal, 255
lógica
  combinatoria, 441
  negativa, 457
  positiva, 456
  secuencial, 441
  unidad, CLU, 441
LRU, 259
LSB, bit menos significativo, 23
LSI, 461

LUT, tabla de búsquedas, en diseño de unidades aritméticas, 196, 254

M

MAC, 356
Macintosh, 107
macroinstrucciones, lenguaje ensamblador, 151, 160, 175-177
maestro-esclavo, 472
magnitud signada, 31
mantisa, en números de punto flotante, 36
máquina
código de, 8
lenguaje de, 8, 99
matrices de compuertas programables por el usuario, 491
matriz totalmente asociativa 262
Mauchly, John, 3
MCB, 326
medio
almacenamiento, 320
control de acceso al, 356
mejor lugar libre, 277
memoria
aleatoria de doble puerto, 285
aleatoria de lectura dual, 285
de lectura, ROM, 11, 252
dirección de definición, 6
direccionable por contenido, 259, 281
entrada-salida mapeada en, 103
en ARC, 109
flash, 253
jerarquías, 243-244
locación, 101
mapa, 102
no volátil, 320
unidad de administración, 172
unidad de definición, 4
RAM estática, 245
video asignado a, 340
memorias direccionables por contenido, 259, 281
menos frecuentemente usado, LFU, 259
menos recientemente usado, LRU, 259
Merced, 387, 411-414
métodos, Java, 140
MFM, 321
microarquitectura, 192
microcontrolador, 203
definición, 10
microinstrucción, 201
microprocesador, 11
microprograma, 10
MIMD, 417
minuendo, 65
MISD, 418
MMU, 172
MMX, 177-185
instrucciones, 177
modelo von Neumann, 4, 434
módem, 347
modo
indirecto de direccionamiento, 129
inmediato de direccionamiento, 129
supervisor, 218
modos de direccionamiento
basado en registro, 129, 130
direccionamiento base, 154
directo, 129
indexado a través de registro, 129, 130
indexado basado en registro, 129, 130
indirecto a través de registro, 129
inmediato, 129
modulación
en fase, 349
en frecuencia modificada, 321
en frecuencia, 348
por impulsos codificados, 349
en telecomunicaciones, 348
módulo, 85
de carga, 168
objeto, 168
monitor, video, 12, 339
Motorola,
68000, *véase* 68000, Motorola
AltiVec, 177-185
PowerPC, 179, 406-409
MSB, bit más significativo, 23
MSI, 461
multidifusión, 373
multiplexado por división de tiempos, 378
multiplexor, 378, 461
multiplicador matricial, 81
multiprocesador

de memoria compartida, 268
simétrico, 307
MUX, 461

N

nanoalmacenamiento, 218
nemotécnico, esquema, 9, 110
nibble, 101
Nintendo, 434
NISDN, 377
niveles (arquitectura de computadoras), 1
niveles de máquina, 6
NNI, 381
no retorno a cero, 321
nodo, 376
norma de video NTSC, 341
norma de video PAL, 341
NRZ, 321
número
conversión
entre sistemas, 23
método de las multiplicaciones, 25
método de los restos, 23
representación,
decimal codificado en binario, 34
en exceso, 33
representaciones,
complemento a dos, 32
complemento a uno, 32
magnitud y signo, 31
números en complemento a dos, 32

O

octeto, 101
operaciones de copia de vectores, 178
ópticos
discos, 330
ordenamiento por burbujeo, 378
OSI, 354

P

página
falla, 271
marco, 271
tabla, 272
paginación, 271
paginación por demanda, 273
palabra, 101
cuádruple, 116
de instrucción muy larga, VLIW, 387
doble, 116
rotulada, formato de datos de SPARC, 116
en cintas magnéticas, 328
palanca de control, 337
pantalla sensible al tacto, 137, 336
paquete, 354
paralelo,
procesamiento, 387, 414
tiempo, 415
paridad impar, 362
particiones, en CM-5, 432
Pascal, Blas, 1
Pascalina, 1
PC, 112
PCI, 308
PCM, 349
PDP-4, 145
PDP-8, 145
peine, disco magnético, 321
Pentium, Intel, 178
II, 13
período, 475
persistencia indefinida, del medio magnético, 320
pila
marco, 130, 133
puntero, 113, 132
pila de protocolo, 354
píxel, 340
PLA, 467
placa madre, 12, 13, 301, 434
planos, en discos compactos, 330
plato, disco magnético, 320
PM, 349
polinómico,
código, 368
método, 24
polinomio generador, 368
política
de escritura diferida, *cache*, 264
de escritura inmediata, *cache*, 264
de escritura sin asignación, *cache*, 264
de escritura y asignación, *cache*, 264

de reemplazo óptimo, 259
POS, 455, 499
PostScript, 339
postulados de Huntington, 451
PowerPC, 107, 179, 406-409
precisión,
en números de punto fijo, 20
PRI, 377
primer lugar libre, 277
primero que entra, primero que sale, 259
principio
de dualidad, 450
de localidad, 254
prioridad de pérdida de celda, en paquetes ATM, 381
procedimiento, 130
procesadores vectoriales, 177
producto de sumas, 455, 499
profundidad de circuito, 519
programa de enlace, 168
programable
generador de sonidos, 434
matriz lógica, 466
PROM, 253
programador de, 252, 436
propagación, en arrastre anticipado, 77
propiedad
conmutativa del álgebra de Boole, 450
distributiva del álgebra de Boole, 450
protección, 275
PSG, generador programable de sonidos, 435
PSR, 112
PTI, en paquetes ATM, 381
puente, 307, 308, 359, 376
de enlace, 359
puerto, 374
gráfico avanzado, 307
puertos estándar, 374
puntero, 335
a ventana actual, CWP, 400
punto de control de señalización, 381
punto flotante
números, 35-48
bit implícito, 37
definición, 19
mantisa, 36
precisión, 36
rango, 35
unidad, 406

Q

Quine-McCluskey, método de, 509
QWERTY, 334

R

radiofrecuencia, 287
RAM, 245
Rambus, Inc. 285
rango
en números de punto fijo, 21
en números de punto flotante, 35
ratón, 335
recompilación, 107
red
de área local, LAN, 347, 354
de Clos, 421
de encaminamiento automático, 378
de interconexión, 418
digital de servicios integrados, ISDN, 379
no bloqueante, reconfiguración, 421
redes de área extendida, WAN, 347
reducción
algebraica, 499
por mapas de Karnaugh, 499
tabular, 499
reducido
computadora con conjunto de instrucciones, 108
tabla de opciones, 513
redundantes, 90
referencia previa, 162, 164
registro, 488
archivo de, 400
bloque de, 105
cinta, 328
de enlace, 113, 133
de estado del procesador, 112
direccionamiento basado en, 129, 130
direccionamiento

indexado basado en, 129, 130
direccionamiento indexado, 129, 130
direccionamiento indirecto, 129
físico, cinta, 329
lenguaje de transferencia, 219
lógico, cinta, 329
problema de asignación, 404
ventana, 387
repetidor, 359
resolvedor, 376
respuesta, 415, 416
resta binaria, 65
restador serie, 65, 66
retardado
carga, 396
salto, 396
reubicación, 171
de programas, 168
diccionario, 168
riesgo, 474
ringing, 414
ROM, 11, 252
RTL, 219

S

S/360, 145
salto
predicación, en la arquitectura IA-64, 413
tabla, 216
unidad de procesamiento, 406
SCP, 381
SCSI, 308, 324
SEC, código de corrección de errores simples, 365
Sega, 434
Génesis, 434, 435
segmentación, 274, 387, 392
en SPARC, 123
segmento, 274
semántico
análisis, en compilación, 152
brecha, 388
semisumador, 469
seudo operaciones, 121-122
SIMD, 416
procesadores, 177
simulaciones VHDL
controladas por eventos, 228
representación en exceso, 33
sincrónico,
bus, 302
modo de transferencia, 378
sincronismo
de direcciones de columna, CAS, 249
de direcciones de fila, RAS, 249
sincronización, disco magnético, 327
sintetizador, 435
SISD, 416
Sistema
de numeración modular, 63
de numeración posicional, 22
SMP, 307
Sony, 434
SOP, 453, 499
SPARC, 108
segmentación, 123
SRAM, 245
SSI, 461
STM, 378
subred, 359
suma de productos, 453, 499
sumador
completo, 469
con arrastre anticipado, 96
serie, 65
superficie, disco magnético, 320
superposiciones, 270
sustraendo, 65
Swift, Jonathan, 102

T

tabla
de símbolos 162, 164
de verdad, 443
reducida de opciones, 513
tablas de transiciones, 530
tableta digitalizadora, 335
TDM, 378
teclado, 12, 333
Dvorak, 334
tecnología de líneas de transmisión, 287
telar de Jacquard, 3
teorema
chino de los restos, 86
de DeMorgan, 451
del cero, 451

del consenso, 451
del uno, 451
término
máximo, 455
mínimo, 454
términos, producto, 453
Texas Instruments, 435
TI, Texas Instruments, 435
tiempo
de búsqueda, 323
de transferencia, 323
efectivo de acceso, 265
TLB, 278
toggle, 528
topología en estrella, 357
trackball, 335
traducción del lenguaje C
arreglos, 156
buffer de, 278
control de secuencia, 157-160
estructuras, 155
variables automáticas, 154
variables estáticas, 154
variables globales, 154
variables locales, 154
proceso de (compilación), 108
trama, 339
transferencia entre celdas, 353
transistor, 10, 447
trap, 216-218,
trayecto de datos, 104, 192
traza, conexiones, 287, 301
Turing, Alan, 3

U

umbral, para compuertas lógicas, 448
un lenguaje de programación de hardware, 219
UNI, en paquetes ATM, 381
Unicode, 52
unidad
aritmético-lógica, *véase* ALU
central de proceso, *véase* CPU, 4
de control microprogramada, 192
de enteros, 406
de entrada, 4
de salida, computadora, 4
unidades de control
cableadas, 192, 219
UNIVAC, 144

V

variable ingresada en el mapa, 508
variación transitoria, 474
VAX, 145
VCI, en paquetes ATM, 381
ventana, registro, en SPARC, 401
ventanas de registros superpuestos, 387, 398, 399
VHDL, 219, 227
VHSIC, lenguaje de descripción de hardware, 219
Viajes de Gulliver, 102
videojuego hogareño, 433
virtual,
identificador de canal, 381
identificador de trayectoria, en paquetes ATM, 381
máquina, 107
memoria, 270
VLIW, palabra de instrucción muy larga, 387, 410
VLSI, 283, 461
VPI, en paquetes ATM, 381

W

Wilkes, Maurice, 4
WORM, escritura única, muchas lecturas, 332

X

Xeon, 307

Y

Yamaha, 435, 436

Z

Zilog Z 80, 145, 435, 436
zona
disco, 322
internet, 376

Pearson Educación

Visítenos en **www.pearsonedlatino.com**

Argentina
Av. Regimiento de los Patricios 1959
(C1266AAF) Buenos Aires
Argentina
Tel. (54-11) 4309-6100
Fax (54-11) 4309-6199
E-mail: info@pearsoned.com.ar

América Central-Panamá
Barrio La Guaria, Moravia
75 metros norte,
Del Portón Norte del Club La Guaria
San José, Costa Rica
Tel. (506) 235 72 76
Fax. (506) 297 28 52
E-mail: envwong@racsa.co.cr

Brasil
Rua Emilio Goeldi 747, Lapa
(05065-110) São Paulo - SP
Brasil
Tel. (5511) 36111-0201
Fax (5511) 36111-0654

Caribe
Monte Mall, 2do. piso, suite 21-B
Av. Muñoz Rivera
Hato Rey
Puerto Rico 00918-4261
Tel. (787) 751-4830
Fax (787) 751-1677
E-mail: awlcarib@caribe.net
y awlcarib@caribe.net

Chile
Av Manuel Montt 1452
Providencia
Santiago, Chile
Tel. (562) 269 2089
Fax (562) 274 6158
E-mail: infopear@pearsoned.cl

Colombia
Carrera 68 # 22-55
Santa Fé de Bogotá, D.C.
Colombia
Tel. (571) 405-9300
Fax (571) 405-9330

España
Nuñez de Balboa 120
(28006) Madrid
España
Tel. (3491) 590-3432
Fax (3491) 590-3448

México
Calle Cuatro No. 25 2do. piso
Fracc. Industrial Alce Blanco
(53370) Naucalpan de Juárez
Estado de México
Tel. (525) 3870700
Fax. (525) 3870811

Uruguay
Casa Juana de América
Av. 8 de Octubre 3061
(11600), Montevideo
Uruguay
Tel./Fax (5982) 486-1617