

ARQUITECTURA DE COMPUTADORAS

DE LOS MICROPROCESADORES A LAS SUPERCOMPUTADORAS

Behrooz Parhami

The book cover features a vibrant lime green background. A large, dark silhouette of a human head in profile, facing right, is the central focus. The interior of the head is filled with a complex, layered pattern of circuit board traces in shades of brown, tan, and black, creating a sense of depth and technological complexity. The traces form various shapes, including what looks like a smaller head profile within the main one, suggesting a recursive or self-referential theme. In the bottom left corner, the McGraw Hill logo is visible, consisting of the words 'Mc', 'Graw', and 'Hill' stacked vertically in white text on a red rectangular background.

**Mc
Graw
Hill**

ARQUITECTURA DE COMPUTADORAS

DE LOS MICROPROCESADORES
A LAS SUPERCOMPUTADORAS

Behrooz Parhami

University of California, Santa Barbara

Revisión técnica

Alejandro Velázquez Mena

Universidad Nacional Autónoma de México



MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MADRID • NUEVA YORK
SAN JUAN • SANTIAGO • AUCKLAND • LONDRES • MILÁN • MONTREAL • NUEVA DELHI
SAN FRANCISCO • SÃO PAULO • SINGAPUR • SAN LUIS • SIDNEY • TORONTO

Director Higher Education: Miguel Ángel Toledo Castellanos

Director editorial: Ricardo A. del Bosque Alayón

Editor sponsor: Pablo Eduardo Roig Vázquez

Editora de desarrollo: Ana Laura Delgado Rodríguez

Supervisor de producción: Zeferino García García

Traductor: Víctor Campos Olguín

ARQUITECTURA DE COMPUTADORAS.

De los microprocesadores a las supercomputadoras

Prohibida la reproducción total o parcial de esta obra,
por cualquier medio, sin la autorización escrita del editor.



DERECHOS RESERVADOS © 2007 respecto a la primera edición en español por
McGRAW-HILL/INTERAMERICANA EDITORES, S.A. DE C.V.

A Subsidiary of The McGraw-Hill Companies, Inc.

Edificio Punta Santa Fe

Prolongación Paseo de la Reforma 1015, Torre A

Piso 17, Colonia Desarrollo Santa Fe,

Delegación Álvaro Obregón

C.P. 01376, México, D. F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

ISBN-13: 978-970-10-6146-6

ISBN-10: 970-10-6146-2

Traducido de la primera edición de la obra: *Computer Architecture: From Microprocessors To Supercomputers*
by Behrooz Parhami.

Copyright © 2005 by Oxford University Press, Inc. All rights reserved.

ISBN-13: 978-0-19-515455-9

ISBN: 0-19-515455-X

1234567890

09865432107

Impreso en México

Printed in Mexico

UN VISTAZO A LA ESTRUCTURA

		Partes	Capítulos
C P U		1. Antecedentes y motivación	1. Circuitos digitales combinacionales 2. Circuitos digitales con memoria 3. Tecnología de sistemas de computación 4. Rendimiento de computadoras
		2. Arquitectura de conjunto de instrucciones	5. Instrucciones y direccionamiento 6. Procedimientos y datos 7. Programas en lenguaje ensamblador 8. Variaciones en el conjunto de instrucciones
		3. La unidad aritmética/lógica	9. Representación de números 10. Sumadores y ALU simples 11. Multiplicadores y divisores 12. Aritmética de punto flotante
		4. Ruta de datos y control	13. Pasos de ejecución de instrucciones 14. Síntesis de unidad de control 15. Ruta de datos encauzadas 16. Límites del rendimiento de <i>pipeline</i>
		5. Diseño de sistemas de memoria	17. Conceptos de memoria principal 18. Organización de memoria caché 19. Conceptos de memoria masiva 20. Memoria virtual y paginación
		6. Entrada/salida e interfases	21. Dispositivos de entrada/salida 22. Programación de entrada/salida 23. Buses, ligas e interfases 24. Conmutación conceptual e interrupciones
		7. Arquitecturas avanzadas	25. Hacia un mayor rendimiento 26. Procesamientos vectorial y matricial 27. Multiprocesamiento lento de memoria compartida 28. Multicomputación distribuida

CONTENIDO

Prefacio xv

■ **PARTE UNO • ANTECEDENTES Y MOTIVACIÓN 1**

1. Circuitos digitales combinacionales 3

- 1.1 Señales, operadores lógicos y compuertas 3
- 1.2 Funciones y expresiones booleanas 7
- 1.3 Diseño de redes de compuertas 8
- 1.4 Partes combinacionales útiles 11
- 1.5 Partes combinacionales programables 13
- 1.6 Estimaciones de temporización y de circuito 15

Problemas 17

Referencias y lecturas sugeridas 19

2. Circuitos digitales con memoria 21

- 2.1 Latches, flip-flops y registros 21
- 2.2 Máquinas de estado finito 24
- 2.3 Diseño de circuitos secuenciales 25
- 2.4 Partes secuenciales útiles 28
- 2.5 Partes secuenciales programables 31
- 2.6 Relojes y temporización de eventos 32

Problemas 34

Referencias y lecturas sugeridas 37

3. Tecnología de sistemas de computación 38

- 3.1 De los componentes a las aplicaciones 39
- 3.2 Sistemas de cómputo y sus partes 41
- 3.3 Generaciones de progreso 45
- 3.4 Tecnologías de procesador y memoria 48
- 3.5 Periféricos, I/O y comunicaciones 51
- 3.6 Sistemas de software y aplicaciones 54

Problemas 56

Referencias y lecturas sugeridas 58

4. Rendimiento de computadoras 59

- 4.1 Costo, rendimiento y costo/rendimiento 59
- 4.2 Definición de rendimiento de computadora 62

- 4.3 Mejora del rendimiento y ley de Amdahl 65
- 4.4 Medición del rendimiento contra modelado 67
- 4.5 Informe del rendimiento de computadoras 72
- 4.6 Búsqueda de mayor rendimiento 74
- Problemas 76**
- Referencias y lecturas sugeridas 80**

■ PARTE DOS • ARQUITECTURA DE CONJUNTO DE INSTRUCCIONES 81

5. Instrucciones y direccionamiento 83

- 5.1 Visión abstracta del hardware 83
- 5.2 Formatos de instrucción 86
- 5.3 Aritmética simple e instrucciones lógicas 89
- 5.4 Instrucciones *load* y *store* 91
- 5.5 Instrucciones *jump* y *branch* 93
- 5.6 Modos de direccionamiento 97
- Problemas 99**
- Referencias y lecturas sugeridas 102**

6. Procedimientos y datos 103

- 6.1 Llamadas de procedimiento simples 103
- 6.2 Uso de la pila para almacenamiento de datos 106
- 6.3 Parámetros y resultados 108
- 6.4 Tipos de datos 110
- 6.5 Arreglos y apuntadores 113
- 6.6 Instrucciones adicionales 116
- Problemas 120**
- Referencias y lecturas sugeridas 122**

7. Programas en lenguaje ensamblador 123

- 7.1 Lenguajes de máquina y ensamblador 123
- 7.2 Directivas de ensamblador 126
- 7.3 Seudoinstrucciones 127
- 7.4 Macroinstrucciones 130
- 7.5 Ligado y cargado 131
- 7.6 Corrida de programas ensamblador 133
- Problemas 136**
- Referencias y lecturas sugeridas 138**

8. Variaciones en el conjunto de instrucciones 139

- 8.1 Instrucciones complejas 139
- 8.2 Modos de direccionamiento alterno 141
- 8.3 Variaciones en formatos de instrucción 145

- 8.4 Diseño y evolución del conjunto de instrucciones 147
- 8.5 La dicotomía RISC/CISC 148
- 8.6 Dónde dibujar la línea 151
- Problemas 154**
- Referencias y lecturas sugeridas 156**

■ **PARTE TRES • LA UNIDAD ARITMÉTICA/LÓGICA 157**

9. Representación de números 159

- 9.1 Sistemas numéricos posicionales 159
- 9.2 Conjuntos de dígitos y codificaciones 162
- 9.3 Conversión de base numérica 165
- 9.4 Enteros con signo 166
- 9.5 Números de punto fijo 169
- 9.6 Números de punto flotante 171

Problemas 174

Referencias y lecturas sugeridas 176

10. Sumadores y ALU simples 178

- 10.1 Sumadores simples 178
- 10.2 Redes de propagación de acarreo 180
- 10.3 Conteo e incremento 183
- 10.4 Diseño de sumadores rápidos 185
- 10.5 Operaciones lógicas y de corrimiento 188
- 10.6 ALU multifunción 191

Problemas 193

Referencias y lecturas sugeridas 196

11. Multiplicadores y divisores 197

- 11.1 Multiplicación corrimiento-suma 197
- 11.2 Multiplicadores de hardware 201
- 11.3 Multiplicación programada 204
- 11.4 División corrimiento-resta 206
- 11.5 Divisores de hardware 210
- 11.6 División programada 213

Problemas 215

Referencias y lecturas sugeridas 218

12. Aritmética con punto flotante 219

- 12.1 Modos de redondeo 219
- 12.2 Valores y excepciones especiales 224
- 12.3 Suma con punto flotante 226
- 12.4 Otras operaciones con punto flotante 229

12.5 Instrucciones de punto flotante 230

12.6 Precisión y errores resultantes 233

Problemas 237

Referencias y lecturas sugeridas 239

■ **PARTE CUATRO • RUTA DE DATOS Y CONTROL 241**

13. Pasos de ejecución de instrucciones 243

13.1 Un pequeño conjunto de instrucciones 244

13.2 La unidad de ejecución de instrucciones 246

13.3 Una ruta de datos de ciclo sencillo 247

13.4 Bifurcación y saltos 249

13.5 Derivación de las señales de control 250

13.6 Rendimiento del diseño de ciclo sencillo 253

Problemas 255

Referencias y lecturas sugeridas 257

14. Síntesis de unidad de control 258

14.1 Implementación multiciclo 258

14.2 Ciclo de reloj y señales de control 261

14.3 La máquina de control de estado 264

14.4 Rendimiento del diseño multiciclo 266

14.5 Microprogramación 267

14.6 Excepciones 271

Problemas 273

Referencias y lecturas sugeridas 276

15. Rutas de datos encauzadas 277

15.1 Conceptos de *pipelining* 277

15.2 Atascos o burbujas encauzadas 281

15.3 Temporización y rendimiento encauzado 284

15.4 Diseño de rutas de datos encauzadas 286

15.5 Control encauzado 289

15.6 *Pipelining* óptimo 291

Problemas 293

Referencias y lecturas sugeridas 296

16. Límites del rendimiento de *pipeline* 297

16.1 Dependencias y riesgos de datos 297

16.2 Adelantamiento de datos 300

16.3 Riesgos de la bifurcación *pipeline* 302

16.4 Predicción de bifurcación 304

16.5 *Pipelining* avanzado 306

16.6 Excepciones en una *pipeline* 309

Problemas 310

Referencias y lecturas sugeridas 313

■ **PARTE CINCO • DISEÑO DE SISTEMAS DE MEMORIA 315**

17. Conceptos de memoria principal 317

17.1 Estructura de memoria y SRAM 317

17.2 DRAM y ciclos de regeneración 320

17.3 Impactar la pared de memoria 323

17.4 Memorias encauzada e interpolada 325

17.5 Memoria no volátil 327

17.6 Necesidad de una jerarquía de memoria 329

Problemas 331

Referencias y lecturas sugeridas 334

18. Organización de memoria caché 335

18.1 La necesidad de una caché 335

18.2 ¿Qué hace funcionar a una caché? 338

18.3 Caché de mapeo directo 341

18.4 Caché de conjunto asociativo 342

18.5 Memoria caché y principal 345

18.6 Mejoramiento del rendimiento caché 346

Problemas 348

Referencias y lecturas sugeridas 352

19. Conceptos de memoria masiva 353

19.1 Fundamentos de memoria de disco 354

19.2 Organización de datos en disco 356

19.3 Rendimiento de disco 359

19.4 *Caching* de disco 360

19.5 Arreglos de discos y RAID 361

19.6 Otros tipos de memoria masiva 365

Problemas 367

Referencias y lecturas sugeridas 370

20. Memoria virtual y paginación 371

20.1 Necesidad de la memoria virtual 371

20.2 Traducción de dirección en memoria virtual 373

20.3 *Translation lookaside buffer* 376

20.4 Políticas de sustitución de página 379

20.5 Memorias principal y masiva 382

20.6 Mejora del rendimiento de la memoria virtual 383

Problemas 386
Referencias y lecturas sugeridas 389

■ PARTE SEIS • ENTRADA/SALIDA E INTERFASES 391

21. Dispositivos de entrada/salida 393

- 21.1 Dispositivos y controladores de entrada/salida 393
- 21.2 Teclado y ratón 395
- 21.3 Unidades de presentación visual 397
- 21.4 Dispositivos de entrada/salida de copia impresa 400
- 21.5 Otros dispositivos de entrada/salida 404
- 21.6 Redes de dispositivos de entrada/salida 406

Problemas 408

Referencias y lecturas sugeridas 410

22. Programación de entrada/salida 411

- 22.1 Rendimiento I/O y *benchmarks* 411
- 22.2 Direccionamiento entrada/salida 413
- 22.3 I/O calendarizado: sondeo 416
- 22.4 I/O con base en petición: interrupciones 417
- 22.5 Transferencia de datos I/O y DMA 418
- 22.6 Mejora del rendimiento I/O 421

Problemas 425

Referencias y lecturas sugeridas 428

23. Buses, ligas e interfaces 429

- 23.1 Ligas intra e intersistema 429
- 23.2 Buses y su presentación 433
- 23.3 Protocolos de comunicación de bus 435
- 23.4 Arbitraje y rendimiento de bus 438
- 23.5 Fundamentos de interfaces 440
- 23.6 Estándares en la creación de interfaces 441

Problemas 445

Referencias y lecturas sugeridas 448

24. Conmutación contextual e interrupciones 449

- 24.1 Peticiones al sistema por I/O 449
- 24.2 Interrupciones, excepciones y trampas 451
- 24.3 Manejo de interrupciones simples 453
- 24.4 Interrupciones anidadas 456
- 24.5 Tipos de conmutación contextual 458
- 24.6 Hilos y multihilos 460

Problemas 462

Referencias y lecturas sugeridas 464

■ **PARTE SIETE • ARQUITECTURAS AVANZADAS 465**

25. Hacia un mayor rendimiento 467

- 25.1 Tendencias de desarrollo pasadas y actuales 467
- 25.2 Extensiones ISA impulsadas por rendimiento 470
- 25.3 Paralelismo a nivel de instrucción 473
- 25.4 Especulación y predicción del valor 476
- 25.5 Aceleradores de hardware de propósito especial 479
- 25.6 Procesamientos vectorial, matricial y paralelo 482

Problemas 485

Referencias y lecturas sugeridas 488

26. Procesamiento vectorial y matricial 490

- 26.1 Operaciones sobre vectores 491
- 26.2 Implementación de procesador vectorial 493
- 26.3 Rendimiento del procesador vectorial 497
- 26.4 Sistemas de control compartido 499
- 26.5 Implementación de procesador matricial 501
- 26.6 Rendimiento de procesador matricial 503

Problemas 504

Referencias y lecturas sugeridas 507

27. Multiprocesamiento de memoria compartida 508

- 27.1 Memoria compartida centralizada 508
- 27.2 Cachés múltiples y coherencia de caché 512
- 27.3 Implementación de multiprocesadores simétricos 514
- 27.4 Memoria compartida distribuida 517
- 27.5 Directorios para guía de acceso a datos 519
- 27.6 Implementación de multiprocesadores asimétricos 521

Problemas 524

Referencias y lecturas sugeridas 527

28. Multicomputación distribuida 528

- 28.1 Comunicación mediante paso de mensajes 528
- 28.2 Redes de interconexión 532
- 28.3 Composición y enrutamiento de mensajes 535
- 28.4 Construcción y uso de multicomputadoras 537
- 28.5 Computación distribuida basada en red 540
- 28.6 Computación en retícula y más allá 542

Problemas 543

Referencias y lecturas sugeridas 547

PREFACIO

“...hay una tendencia cuando en realidad comienzas a aprender algo acerca de alguna cosa: no querer escribir respecto de ello sino más bien seguir aprendiendo en relación con ello... a menos que seas muy egoísta, lo cual, desde luego, explica muchos libros.”

Ernest Hemingway, Death in the Afternoon

■ El contexto de la arquitectura de computadoras

La arquitectura de computadoras constituye un área de estudio que se refiere a las computadoras digitales en la interfaz entre hardware y software. Aquella está más orientada al hardware que los “sistemas de cómputo”, un área que usualmente se cubre en los cursos académicos con el mismo nombre en las materias de ciencias o ingeniería de la computación, y se preocupa más por el software que los campos conocidos como “diseño computacional” y “organización de computadoras”. No obstante, la materia es bastante fluida y varía enormemente de un texto o curso a otro, en su orientación y cobertura. Esto último explica, en parte, por qué existen tantos libros diferentes acerca de arquitectura de computadoras y por qué, incluso, otro texto en la materia puede ser útil.

En este contexto, la arquitectura de computadoras abarca un conjunto de ideas fundamentales que se aplican al diseño o comprensión de cualquier computadora, desde los más pequeños microprocesadores anidados que controlan los aparatos electrodomésticos, cámaras y muchos otros dispositivos (por ejemplo, máquinas personales, servidores y complejos sistemas de computadoras) hasta las más poderosas supercomputadoras que se encuentran sólo en (y son costeables por) los grandes centros de datos o en los principales laboratorios científicos. También se ramifica en otras áreas más avanzadas, cada una con su propia comunidad de investigadores, revistas especializadas, simposios y, desde luego, lenguaje técnico. Los diseñadores de computadoras no deben dudar en familiarizarse con todo el campo para ser capaces de usar la variedad de métodos disponibles con el propósito de diseñar sistemas rápidos, eficientes y poderosos. Menos obvio es el hecho de que incluso los simples usuarios de computadoras se pueden beneficiar de las ideas fundamentales y de la apreciación de los conceptos más avanzados en la arquitectura de computadoras.

Un tema común en arquitectura de computadoras consiste en enfrentar su complejidad. Gran parte de ésta surge de nuestro deseo de hacer todo tan rápido como sea posible. Algunas de las técnicas resultantes, como la ejecución predictiva y especulativa, están peleadas con otras metas del diseño de sistemas que incluyen bajo costo, compactidad, ahorro de energía, poco tiempo para entrar al mercado y comprobación. La constante problemática de tales requerimientos conflictivos propicia que la arquitectura de computadoras represente un próspero e interesante campo de estudio. Además de lo anterior, se encuentran las fuerzas opositoras de innovación y compatibilidad con las inversiones en habilidades, sistemas y aplicaciones.

■ Ámbito y características

Este libro de texto, que en realidad constituye una prolongación de las notas de clase que el autor desarrolló y mejoró a lo largo de muchos años, cubre las ideas fundamentales de la arquitectura de computadoras con cierta profundidad y ofrece muchos conceptos avanzados que se pueden seguir en cursos de nivel superior, como los de supercomputadoras, procesamiento paralelo y sistemas distribuidos.

Seis características clave apartan a este libro de los textos introductorios de la competencia acerca de arquitectura de computadoras:

- a) *División del material en capítulos con tamaño adecuado para clases:* En el enfoque docente del autor, una clase representa un módulo autocontenido con vínculos a clases anteriores e indicadores hacia lo que se estudiará en lo futuro. Cada clase, con duración de una o dos horas, tiene un tema o título y procede desde la motivación hacia los detalles y la conclusión.
- b) *Gran cantidad de problemas significativos:* Al final de cada de capítulo se ofrecen al menos 16 problemas, que además de estar bien pensados, muchos de ellos probados en clase, clarifican el material del capítulo, ofrecen nuevos ángulos de vista, vinculan el material del capítulo con temas en otros capítulos o introducen conceptos más avanzados.
- c) *Énfasis tanto en la teoría subyacente como en los diseños reales:* La habilidad para enfrentar la complejidad requiere tanto una profunda comprensión de los puntales de la arquitectura de computadoras como de ejemplos de diseños que ayudan a clarificar la teoría. Estos últimos también proporcionan bloques constructores para síntesis y puntos de referencia con el fin de realizar comparaciones de costo-rendimiento.
- d) *Vinculación de la arquitectura de computadoras con otras áreas de la computación:* La arquitectura de computadoras se nutre con otros subcampos de diseño de sistemas de cómputo. Tales vínculos, desde los obvios (arquitectura de conjunto de instrucciones frente al diseño de compilador) hasta los sutiles (interjuego de la arquitectura con confiabilidad y seguridad), se explican a todo lo largo del libro.
- e) *Amplia cobertura de tópicos importantes:* El texto cubre casi todos los temas fundamentales de la arquitectura de computadoras; en este sentido, ofrece una visión equilibrada y completa de esta área. Los ejemplos de material que no se encuentran en muchos otros textos, incluyen cobertura detallada de aritmética computacional (capítulos 9-12) y computación de alto rendimiento (capítulos 25-28).
- f) *Terminología/notación unificada y consistente:* Se hace gran esfuerzo por usar terminología/notación consistente a lo largo del texto. Por ejemplo, r siempre representa la base de la representación numérica, k el ancho de palabra y c el acarreo (*carry*). De igual modo, los conceptos y estructuras se identifican de manera consistente con nombres únicos bien definidos.

■ Resumen de temas

Las siete partes de este libro, cada una compuesta con cuatro capítulos, se escribieron con las siguientes metas:

La parte 1 establece las etapas, proporciona contexto, repasa algunos de los temas que se consideran como prerrequisito y brinda un adelanto de lo que está por venir en el resto del libro. Se incluyen dos capítulos de repaso acerca de circuitos y componentes digitales, una discusión de tipos de sistemas de cómputo, un panorama de la tecnología de computación digital y una perspectiva detallada en el rendimiento de los sistemas de cómputo.

La parte 2 sitúa la interfase del usuario con hardware de computadora, también conocida como arquitectura con conjunto de instrucciones (ISA, por sus siglas en inglés). En esta tesitura, se describe el conjunto de instrucciones de los miniMIPS (una máquina simplificada, aunque muy realista, para la que existe material de referencia abierta y herramientas de simulación); asimismo, se incluye un capítulo acerca de variaciones en ISA (por ejemplo, RISC frente a CISC) y negociaciones asociadas costo-rendimiento.

Las siguientes dos partes cubren la unidad de procesamiento central (CPU). La parte 3 describe con cierto detalle la estructura de las unidades aritméticas/lógicas (ALU, por sus siglas en inglés). Se incluyen discusiones de representaciones de números fijos y de punto flotante, diseño de sumadores de alta rapidez, operaciones de corrimiento y lógicas, y hardware multiplicadores/divisores. También se discuten aspectos de implementación y peligros de la aritmética de punto flotante.

La parte 4 se dedica a las rutas de datos y circuitos de control que comprenden los CPU modernos. Si se comienza con los pasos de ejecución de instrucciones, en consecuencia se derivan los componentes necesarios y los mecanismos de control. A este material le sigue una exposición de estrategias de diseño de control, el uso de una trayectoria de datos encauzados para mejorar el rendimiento y varias limitaciones de encauzamiento que son resultado de dependencias de datos y control.

La parte 5 plantea los sistemas de memoria. Se describen las tecnologías en uso para las memorias primarias y secundarias, junto con sus fortalezas y limitaciones. Se muestra cómo el uso de memorias caché efectivamente supera la brecha de rapidez entre el CPU y la memoria principal. De igual modo, se explica el uso de memoria virtual para brindar la ilusión de una memoria principal vasta.

La parte 6 trata con tópicos de entrada/salida (input/output, I/O) e interfaces. A una discusión de tecnologías de dispositivos I/O le siguen métodos de programación I/O y los papeles de los *buses* y enlaces (incluidos estándares) en comunicación I/O e interfaces. También se cubre lo que se refiere a elementos de procesos y conmutación contextual, para el manejo de excepciones y la computación multihilos.

La parte 7 introduce arquitecturas avanzadas. Se presenta un panorama de estrategias de mejora del rendimiento, más allá de un simple encauzamiento, y se citan ejemplos de aplicaciones que requieren mayor rendimiento. El libro concluye con estrategias de diseño y ejemplos de arquitecturas con base en procesamiento vectorial o matricial, multiprocesamiento y multicomputación.

■ Indicaciones acerca de cómo usar el libro

Para uso en el salón de clase, los temas en cada capítulo de este texto se pueden cubrir en una clase de 1-2 horas de duración. De acuerdo con su propia experiencia, el autor usó los capítulos principalmente para clases de 1.5 horas, durante dos veces a la semana en un trimestre con duración de diez semanas, y omitió o combinó algunos capítulos para ajustar el material a las 18-20 clases disponibles. Sin embargo, la estructura modular del texto se inclina a sí misma hacia otros formatos de clase, autoestudio o repaso del campo por los practicantes. Para los últimos dos casos, los lectores pueden ver cada capítulo como una unidad de estudio (o sea, para una semana) en lugar de verlos como una clase. En este orden de ideas, todos los temas en cada capítulo se deben concluir antes de avanzar hacia el siguiente capítulo. Sin embargo, si se tienen pocas horas de clase disponibles, entonces se pueden omitir algunas de las subsecciones ubicadas al final de los capítulos o introducirse sólo en términos de motivaciones e ideas clave.

Para cada capítulo se proporcionan problemas de complejidad variable, desde los ejemplos o ejercicios numéricos directos hasta estudios o miniproyectos más demandantes. Estos problemas forman parte integral del libro y no se agregaron como ocurrencia tardía con el propósito de hacer el libro más atractivo para su uso como texto. Se incluyen 491 problemas. Si se suponen dos clases por semana, se puede asignar tarea en forma semanal o quincenal, y cada asignación tendrá la cobertura específica de la respectiva media parte (dos capítulos) o parte completa (cuatro capítulos) de su “título”.

Al final de cada capítulo se mencionan referencias de artículos fundamentales acerca de arquitectura de computadoras, ideas de diseño clave e importantes contribuciones en la investigación actual. Estas referencias ofrecen buenos puntos de partida para realizar estudios a profundidad o para preparar ensayos/proyectos finales. Asimismo, aparecen nuevas ideas en esta área en artículos que se presentan cada año en el International Symposium on Computer Architecture [ISCA]. Otras reuniones técnicas

de interés incluyen el Symposium on High-Performance Computer Architecture [HPCA, por sus siglas en inglés], el International Parallel and Distributed Processing Symposium [IPDP] y la International Conference on Parallel Processing [ICPP]. Las revistas especializadas relevantes incluyen *IEEE Transactions on Computers* [TrCo], *IEEE Transactions on Parallel and Distributed Systems* [TrPD], *Journal of Parallel and Distributed Computing* [JPDC] y *Communications of the ACM* [CACM]. Otros artículos generales y temas de amplio interés aparecen en *IEEE Computer* [Comp], *IEEE Micro* [Micr] y *ACM Computing Surveys* [CoSu].

■ Reconocimientos

Este texto, *Arquitectura de computadoras: De los microprocesadores a las supercomputadoras*, representa una prolongación de las notas de clase que el autor usa para el curso de licenciatura de división superior ECE 154: “Introducción a la arquitectura de computadoras”, en la Universidad de California, Santa Bárbara, y, en forma rudimentaria, en muchas otras instituciones antes de 1988. El texto se benefició enormemente de importantes observaciones, curiosidad y aliento de muchos estudiantes en estos cursos. ¡Extiendo un sincero agradecimiento a todos ellos! Agradezco también a [los editores de la edición original en inglés] Peter Gordon, quien inició el proyecto editorial, a Danielle Christensen, quien lo guió hasta su finalización, y a Karen Shapiro, quien dirigió hábilmente el proceso de edición. Por último, agradezco enormemente el otorgamiento de permiso del Dr. James R. Larus, de Microsoft Research para usar sus simuladores SPIM.

■ Referencias generales y lecturas

La lista que viene a continuación tiene referencias de dos tipos: 1) libros que han influido enormemente al texto actual, y 2) fuentes de referencia general para estudios e investigación a profundidad. Los libros y otros recursos relevantes para capítulos específicos se citan en las listas de referencias al final de los capítulos.

- [Arch] La página WWW de arquitectura de computadoras, recurso Web mantenido por el Departamento de Ciencias de la Computación, Universidad de Wisconsin, Madison, tiene un cúmulo de información acerca de organizaciones, grupos, proyectos, publicaciones, eventos y personas relacionadas con la arquitectura: <http://www.cs.wisc.edu/~arch/www/index.html>
- [CACM] *Communications of the ACM*, revista especializada de la Association for Computing Machinery.
- [Comp] *IEEE Computer*, revista técnica de la IEEE Computer Society.
- [CoSu] *Computing Surveys*, revista especializada de la Association for Computing Machinery.
- [Henn03] Hennessy, J. L. y D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 3a. ed., 2003.
- [HPCA] *Proceedings of the Symposium(s) on High-Performance Computer Architecture*, patrocinado por el IEEE. La 10a. HPCA se realizó del 14 al 18 de febrero de 2004 en Madrid, España.
- [ICPP] *Proceedings of the International Conference(s) on Parallel Processing*, se lleva a cabo anualmente desde 1972. La 33a. ICPP se realizó del 15 al 18 de agosto de 2004 en Montreal, Canadá.
- [IPDP] *Proceedings of the International Parallel and Distributed Processing Symposium(s)*, que se formó en 1998 por la fusión del IPPS (realizado anualmente desde 1987) y el SPDP (realizado anualmente desde 1989). El último IPDPS se realizó del 26 al 30 de abril de 2004 en Santa Fe, Nuevo México.
- [ISCA] *Proceedings of the International Symposium(s) on Computer Architecture*, se organiza anualmente desde 1973, por lo general en mayo o junio. El 31 ISCA se llevó a cabo del 19 al 23 de junio de 2004 en Munich, Alemania.

- [JPDC] *Journal of Parallel and Distributed Computing*, publicado por Academic Press.
- [Micr] *IEEE Micro*, revista técnica publicada por la IEEE Computer Society.
- [Muel00] Mueller, S. M. y W. J. Paul, *Computer Architecture: Complexity and Correctness*, Springer, 2000.
- [Patr98] Patterson, D. A. y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2a. ed., 1998.
- [Rals93] Ralston, A. y E. D. Reilly (eds.), *Encyclopedia of Computer Science*, Van Nostrand Reinhold, 3a. ed., 1993.
- [Siew82] Siewiorek, D. P., C. G. Bell y A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, 1982.
- [Sohi98] Sohi, G. (ed.), *25 Years of the International Symposia on Computer Architecture: Selected Papers*, ACM Press, 1998.
- [Stal03] Stallings, W., *Computer Organization and Architecture*, Prentice Hall, 6a. ed., 2003.
- [Swar76] Swartzlander, E. E., Jr (ed.), *Computer Design Development: Principal Papers*, Hayden, 1976.
- [TrCo] *IEEE Trans. Computers*, revista publicada por la IEEE Computer Society.
- [TrPD] *IEEE Trans. Parallel and Distributed Systems*, revista publicada por IEEE Computer Society.
- [Wilk95] Wilkes, M. V., *Computing Perspectives*, Morgan Kaufmann, 1995.

PARTE UNO

ANTECEDENTES Y MOTIVACIÓN

“La arquitectura es el juego aprendido, correcto y magnífico, de las formas ensambladas en la luz.”
LeCorbusier

“Creo que el futuro arquitecto de computadoras es un arquitecto de sistemas, no simplemente un arquitecto de procesador; de modo que uno debe conjuntar tecnología de software, aplicaciones de sistemas, aritmética, todo en un sistema complejo que tenga un comportamiento estadístico que no sea analizado inmediata o simplemente...”
Michael J. Flynn, viendo hacia adelante, circa 1998

TEMAS DE ESTA PARTE

1. Circuitos digitales combinacionales
2. Circuitos digitales con memoria
3. Tecnología de sistemas de computación
4. Rendimiento de computadoras

La arquitectura de computadoras abarca un conjunto de ideas centrales aplicables al diseño o comprensión de virtualmente cualquier computadora digital, desde los más pequeños sistemas anidados hasta las más grandes supercomputadoras. La arquitectura de computadoras no es sólo para diseñadores de computadoras; incluso los simples usuarios se benefician de un firme asidero de las ideas centrales y de una apreciación de los conceptos más avanzados en este campo. Ciertas realizaciones clave, como el hecho de que un procesador de $2x$ GHz no necesariamente es el doble de rápido que un modelo de x GHz, requieren una capacitación básica en arquitectura de computadoras.

Esta parte comienza con la revisión de los componentes de hardware usados en el diseño de circuitos digitales y subsistemas. Los elementos combinacionales, incluidos compuertas, multiplexores, demultiplexores, decodificadores y codificadores, se cubren en el capítulo 1, mientras que los circuitos secuenciales, ejemplificados por archivos de registro y contadores, constituyen el tema del capítulo 2. En el capítulo 3, se presenta un panorama de los desarrollos en tecnología de computación y su estado actual. A esto sigue, en el capítulo 4, una discusión del rendimiento absoluto y relativo de los sistemas de cómputo, acaso el capítulo individual más importante, pues monta el escenario para los métodos de mejoramiento de rendimiento que se presentarán a lo largo del resto del libro.

■ CAPÍTULO 1

CIRCUITOS DIGITALES COMBINACIONALES

“Solíamos pensar que si conocíamos uno, conocíamos dos, porque uno y uno son dos. Ahora encontramos que debemos aprender muchas más cosas acerca del ‘y’.”

Sir Arthur Eddington

“Este producto contiene minúsculas partículas cargadas eléctricamente, que se mueven a velocidades superiores a los 500 millones de millas por hora. Manéjese con extremo cuidado.”

Propuesta de etiqueta de advertencia acerca de la verdad en el producto, colocada en todos los sistemas digitales (fuente desconocida)

TEMAS DEL CAPÍTULO

- 1.1 Señales, operadores lógicos y compuertas
- 1.2 Funciones y expresiones booleanas
- 1.3 Diseño de redes de compuertas
- 1.4 Partes combinacionales útiles
- 1.5 Partes combinacionales programables
- 1.6 Estimaciones de temporización y de circuito

El interés por estudiar arquitectura de computadoras requiere de familiaridad con el diseño digital; en este sentido, se supone que el lector la tiene. La cápsula de revisión que se presenta en éste y el capítulo siguiente tienen la intención de refrescar la memoria del lector y ofrecerle una base de apoyo para que comprenda la terminología y los diseños en el resto del libro. En este capítulo se revisarán algunos conceptos clave de los circuitos digitales combinacionales (sin memoria), también se introducen algunos componentes útiles que se encuentran en muchos diagramas de este libro. Los ejemplos incluyen buffers tres estados (regulares o de inversión), multiplexores, decodificadores y codificadores. Esta revisión continúa en el capítulo 2, que trata acerca de los circuitos digitales secuenciales (con memoria). Los lectores que tengan problemas para comprender el material de estos dos capítulos, deberán consultar cualquiera de los libros de diseño lógico que se incluyen como referencias al final del capítulo.

■ 1.1 Señales, operadores lógicos y compuertas

Todos los elementos de información en las computadoras digitales, incluidas las instrucciones, números y símbolos, se codifican como señales electrónicas que casi siempre tienen *dos valores*. Aun cuando las *señales multivaluadas* y los circuitos lógicos asociados son accesibles y se usan ocasionalmente,

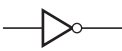



Nombre	NOT	AND	OR	XOR
Símbolo gráfico				
Signo de operador y alternativa(s)	x' $\neg x$ o \bar{x}	xy $x \wedge y$	$x \vee y$ $x + y$	$x \oplus y$ $x \neq y$
Salida 1 ssi (si y sólo si) 1:	Entrada es 0	Ambas entradas son 1	Al menos una entrada es 1	Entradas diferentes
Expresión aritmética	$1 - x$	$x \times y$ o xy	$x + y - xy$	$x + y - 2xy$

Figura 1.1 Algunos elementos básicos de los circuitos lógicos digitales. Se destacan los signos de operador usados en este libro.

las computadoras digitales modernas son predominantemente binarias. Las *señales binarias* se pueden representar mediante la presencia o ausencia de alguna propiedad eléctrica como voltaje, corriente, campo o carga. En este libro los dos valores de una señal binaria se referirán como “0” o “1”. Estos valores pueden representar los dígitos de un número con base 2 en la forma natural o se usan para denotar estados (encendido/apagado), condiciones (falso/verdadero), opciones (trayecto A/trayecto B), etc. La asignación de 0 y 1 a estados binarios o condiciones es arbitraria, pero es más común que el 0 represente “apagado” o “falso” y el 1 corresponda a “encendido” o “verdadero”. Cuando las señales binarias se representen mediante voltaje alto/bajo, la asignación de voltaje alto a 1 conduce a *lógica positiva* y lo opuesto se considera como *lógica negativa*.

Los operadores lógicos representan abstracciones para especificar transformaciones de señales binarias. Existen $2^2 = 4$ posibles operadores de entrada sencilla, porque la tabla de verdad de determinado operador tiene dos entradas (correspondientes a que la entrada sea 0 o 1) y cada entrada se puede llenar con 0 o 1. Un operador de dos entradas con entradas binarias se puede definir en $2^4 = 16$ formas diferentes, dependiendo de si produce una salida 0 o 1 para cada una de las cuatro posibles combinaciones de valores de entrada. La figura 1.1 muestra el operador de entrada sencilla conocido como NOT (*complemento* o *inversor*) y tres de los operadores de dos entradas de uso más común: AND, OR y XOR (OR exclusiva). Para cada uno de estos operadores se proporciona el signo que se usa en las expresiones lógicas y la forma alterna que favorecen los libros de diseño lógico. En la figura 1.1 se destacan los signos de operador usados en dichos libros. En particular, en este texto se usará para OR “ \vee ” en lugar del más común “+”, pues también se tratará mucho acerca de la suma y, de hecho, a veces la suma y OR se usarán en el mismo párrafo o diagrama. Por otra parte, para AND la simple yuxtaposición de los operandos no propiciará ningún problema, pues AND es idéntico a multiplicación para las señales binarias.

La figura 1.1 también relaciona los operadores lógicos con operadores aritméticos. Por ejemplo, complementar o invertir una señal x produce $1 - x$. Puesto que tanto AND como OR son *asociativos*, lo que significa que $(xy)z = (yz)x$ y $(x \vee y) \vee z = x \vee (y \vee z)$, dichos operadores se pueden definir con más de dos entradas, sin causar ambigüedad acerca de sus salidas. Además, como consecuencia de que el símbolo gráfico para NOT consiste de un triángulo que representa la operación identidad (o ninguna operación en absoluto) y una pequeña “burbuja” que significa inversión, los diagramas lógicos se pueden representar de manera simple y menos confusa si se ponen burbujas de inversión en las entradas o salidas de las compuertas lógicas. Por ejemplo, una compuerta AND y un inversor conectado a su salida se pueden fusionar en una sola compuerta NAND, que se dibuja como una compuerta AND con una

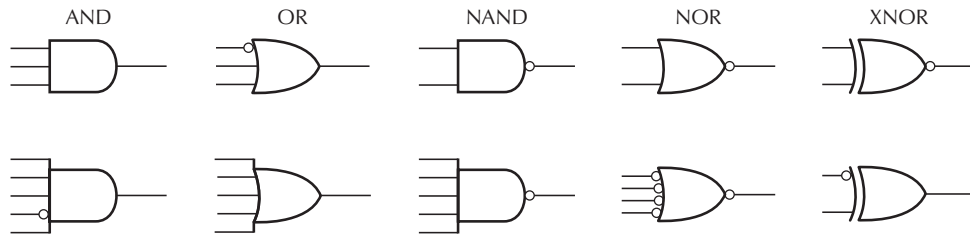


Figura 1.2 Compuertas con más de dos entradas o con señales invertidas en la entrada o salida.

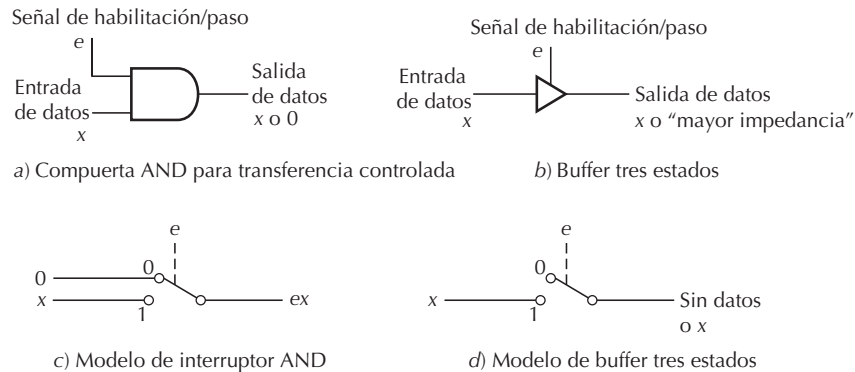


Figura 1.3 Una compuerta AND y un buffer tres estados pueden actuar como interruptores o válvulas controladas. Un buffer de inversión es similar a una compuerta NOT.

burbuja en su línea de salida. De igual modo, las compuertas se pueden definir como NOR y XNOR (figura 1.2). Las burbujas también se pueden colocar en las entradas de una compuerta, ello conduce a la representación gráfica de operaciones como $x' \vee y \vee z$ con un símbolo de compuerta.

Muchas de las variables en un programa reciben nombre, el nombre de una señal lógica se debe escoger con cuidado para expresar información útil acerca del papel de dicha señal. En lo posible se evitarán los nombres o muy cortos o muy largos. A una señal de control cuyo valor es 1 se le refiere como “postulada”, mientras que una señal con valor 0 no es postulada. Postular una señal de control constituye una forma usual de propiciar una acción o evento. Si los nombres de señal se eligen cuidadosamente, una señal denominada “*sub*” quizá provoque que se realice una operación de sustracción cuando se postula, mientras que una señal atada de tres bits “*oper*” codificará la realización de una de ocho posibles operaciones por alguna unidad. Cuando la no postulación de una señal provoca un evento, el nombre de aquella aparecerá en forma de complemento para dar claridad; por ejemplo, la señal *add'*, cuando no está postulada, puede hacer que la suma se realice. También es posible aplicar un nombre, como *add' sub*, para que una señal cause dos diferentes acciones según su valor.

Si una entrada de una compuerta AND de dos entradas se visualiza como señal de control y la otra como señal de datos, una significará que la postulación de la señal de control permite que la señal de datos se propague a la salida, mientras que la no postulación de la señal de control fuerza a la salida a 0, independientemente de los datos de entrada (figura 1.3). Por ende, una compuerta AND puede actuar como un interruptor o válvula de datos controlada por una señal *habilitación e* o *paso*. Un mecanismo alternativo para este propósito, que también se muestra en la figura 1.3, representa un *buffer tres estados* cuya salida es igual a la entrada de datos *x* cuando la señal de control *e* se postula y supone un

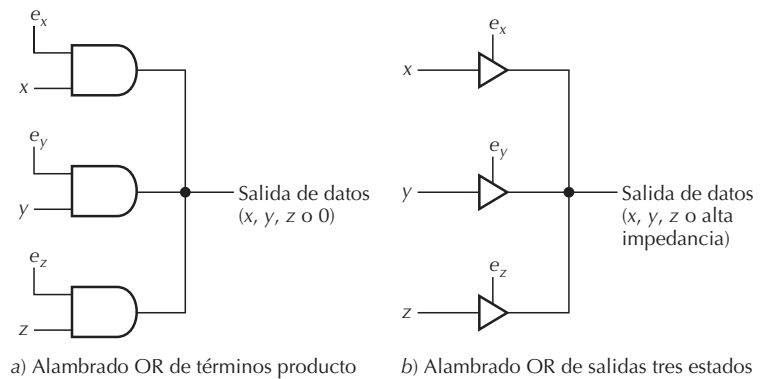


Figura 1.4 El alambrado OR permite unir varias señales controladas.

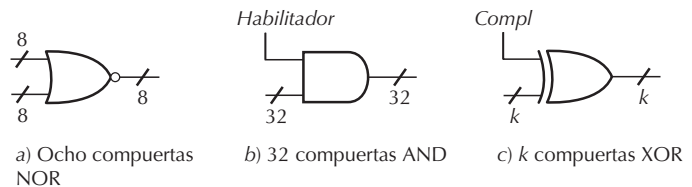


Figura 1.5 Arreglos de compuertas lógicas representadas por el símbolo de una sola compuerta.

valor indeterminado (alta impedancia en términos eléctricos) cuando e no es postulada. Un buffer tres estados aísla efectivamente la salida de la entrada siempre que la señal de control no se postule. Una compuerta XOR con un control y una señal de datos se puede visualizar como un *inversor controlado* que invierte los datos si su control se postula y queda, de otro modo, invariable.

Las salidas de varios interruptores AND, buffer tres estados o de inversión se pueden conectar unos a otros para una función OR implícita o alambrada. De hecho, una aplicación primaria de los buffers tres estados consiste en conectar un número importante de fuentes de datos (como celdas de memoria) a una línea común de datos a través de la que viajan los datos hacia un receptor. En la figura 1.4, cuando sólo una de las señales de habilitación se postula, los datos correspondientes pasan a través y prevalecen en el lado de salida. Cuando no se postula señal de habilitación, entonces la salida será 0 (para compuertas AND) o alta impedancia (para buffers tres estados). Cuando se postula más de una señal de habilitación, la OR lógica de las entradas de datos asociados prevalecen en el lado de salida, aunque esta situación se evita con frecuencia.

De manera recurrente se usa un arreglo de compuertas idénticas para combinar haces de señales. Para bosquejar tal arreglo, en el texto se dibuja sólo una compuerta y se indica, mediante una marca gruesa y un entero junto a ella, cuántas señales o compuertas se involucran. Por ejemplo, la figura 1.5a muestra la operación NOR en lógica de bits realizada en dos haces de ocho bits. Si los haces de entrada son x y y y el haz de salida z , entonces esto equivale a hacer $z_i = (x_i \vee y_i)'$ para cada i . De igual modo, se puede tener un arreglo de 32 interruptores AND, todos unidos a la misma señal *Habilitador*, para controlar el flujo de una palabra de datos de 32 bits desde el lado de entrada hacia el lado de salida (figura 1.5b). Como ejemplo final, se puede usar un arreglo de k compuertas XOR para invertir todos los bits en un haz de k bits siempre que *Compl* se postule (figura 1.5c).

1.2 Funciones y expresiones booleanas

Una señal, que puede ser 0 o 1, representa una *variable booleana*. Una *función booleana* de n variables depende de n variables booleanas y produce un resultado en $\{0, 1\}$. Las funciones booleanas son de interés porque una red de compuertas lógicas con n entradas y una salida implementa una función booleana de n variables. Existen muchas formas para especificar funciones booleanas.

- Una *tabla de verdad* constituye una lista de los resultados de la función para todas las combinaciones de valores de entrada. En este sentido, la tabla de verdad para una función booleana de n variables tiene n columnas de entrada, una columna de salida y 2^n renglones. Asimismo, se le puede usar con m columnas de salida para especificar m funciones booleanas de las mismas variables a la vez (tabla 1.1). Una entrada del tipo *no importa* “x” en una columna de salida significa que el resultado de la función es de interés en dicho renglón, acaso porque dicha combinación de valores de entrada no se espera que surja alguna vez. Una “x” en una columna de entrada significa que el resultado de la función no depende del valor de la variable particular involucrada.
- Una *expresión lógica* se hace con variables booleanas, operadores lógicos y paréntesis. En ausencia de estos últimos, NOT tiene preeminencia sobre AND, que a la vez la tiene sobre OR/XOR. Para una asignación específica de valores a las variables, una expresión lógica se puede evaluar para producir un resultado booleano. Las expresiones lógicas se pueden manipular con el uso de leyes del álgebra booleana (tabla 1.2). Usualmente, la meta de este proceso es obtener una expresión lógica

TABLA 1.1 Tres funciones booleanas de siete variables especificadas en una tabla de verdad compacta con entradas del tipo *no importa* en las columnas de entrada y salida.

Línea #	Siete entradas							Tres salidas		
	$s_{palanca}$	c_{25}	c_{10}	a_{chicle}	a_{barra}	p_{chicle}	p_{barra}	$r_{monedas}$	r_{chicle}	r_{barra}
1	0	x	x	x	x	x	x	0	0	0
2	1	0	0	x	x	x	x	x	0	0
3	1	0	1	x	x	x	x	1	0	0
4	1	1	0	x	x	x	x	1	0	0
5	1	1	1	x	x	0	0	1	0	0
6	1	1	1	x	x	1	1	1	0	0
7	1	1	1	x	0	0	1	1	0	x
8	1	1	1	x	1	0	1	0	0	1
9	1	1	1	0	x	1	0	1	x	0
10	1	1	1	1	x	1	0	0	1	0

TABLA 1.2 Leyes (identidades básicas) del álgebra booleana.

Nombre de la ley	Versión OR	Versión AND
Identidad	$x \vee 0 = x$	$x \wedge 1 = x$
Uno/Cero	$x \vee 1 = 1$	$x \wedge 0 = 0$
Idempotencia	$x \vee x = x$	$x \wedge x = x$
Inverso	$x \vee x' = 1$	$x \wedge x' = 0$
Conmutativa	$x \vee y = y \vee x$	$xy = yx$
Asociativa	$(x \vee y) \vee z = x \vee (y \vee z)$	$(xy)z = x(yz)$
Distributiva	$x \vee (yz) = (x \vee y)(x \vee z)$	$x(y \wedge z) = (xy) \wedge (xz)$
De DeMorgan	$(x \vee y)' = x' y'$	$(xy)' = x' \vee y'$

equivalente que en alguna forma es más simple o más adecuada para la realización en hardware. Una expresión lógica que se forma al operar OR varios términos AND representa una forma de *suma (lógica) de productos*, por ejemplo, $xy \vee yz \vee zx$ o $w \vee x'yz$. De igual modo, al operar AND varios términos OR conduce a una expresión de *producto de sumas (lógicas)*; por ejemplo, $(x \vee y)(y \vee z)(z \vee x)$ o $w'(x \vee y \vee z)$.

- c) Un *enunciado en palabras* puede describir una función lógica de unas cuantas variables booleanas. Por ejemplo, el enunciado “la alarma sonará si la puerta se abre mientras está activado el sistema de seguridad o cuando el detector de humo se dispare”, corresponde a la función booleana $e_{\text{alarma}} = (s_{\text{puerta}}s_{\text{seguridad}}) \vee d_{\text{humo}}$, que relaciona una señal de habilitación (e) con un par de señales de estado (s) y una señal de detector (d).
- d) Un *diagrama lógico* constituye una representación gráfica de una función booleana que también porta información acerca de su realización en hardware. De acuerdo con lo anterior, derivar un diagrama lógico a partir de cualesquiera tipos de especificación mencionados representa el proceso de *síntesis de circuito lógico*. Al hecho de ir a la inversa, desde un diagrama lógico hacia otra forma de especificación, se le denomina *análisis de circuito lógico*. Además de las compuertas y otros componentes elementales, un diagrama lógico puede incluir cajas de varias formas que representan bloques constructores del tipo estándar o subcircuitos previamente diseñados.

Con frecuencia se usa una combinación de los cuatro métodos precedentes, en un esquema jerárquico, para representar hardware de computadora. Por ejemplo, un diagrama lógico de alto nivel, compuesto de subcircuitos y bloques estándar, puede ofrecer la imagen completa. Cada uno de los elementos del tipo no estándar, que no son lo suficientemente simples para ser descritos mediante una tabla de verdad, expresión lógica o enunciado en palabras, puede, a su vez, especificarse mediante otro diagrama, etcétera.

Ejemplo 1.1: Probar la equivalencia de expresiones lógicas Pruebe que los siguientes pares de expresiones lógicas son equivalentes.

- a) Ley distributiva, versión AND: $x(y \vee z) \equiv (xy) \vee (xz)$
- b) Ley de DeMorgan, versión OR: $(x \vee y)' \equiv x'y'$
- c) $xy \vee x'z \vee yz \equiv xy \vee x'z$
- d) $xy \vee yz \vee zx \equiv (x \vee y)(y \vee z)(z \vee x)$

Solución: Cada parte se prueba con un método diferente para ilustrar el rango de posibilidades.

- a) Use el método de tabla de verdad: forme una tabla de verdad de ocho renglones, que corresponde a todas las posibles combinaciones de valores para las tres variables x, y, z . Observe que las dos expresiones conducen al mismo valor en cada renglón. Por ejemplo, $1(0 \vee 1) = (1 \cdot 0) \vee (1 \cdot 1) = 1$.
- b) Use las sustituciones aritméticas que se muestran en la figura 1.1 para convertir este problema de igualdad lógica en la igualdad algebraica de fácil comprobación $1 - (x + y - xy) = (1 - x)(1 - y)$.
- c) Use el análisis de caso: por ejemplo, derive formas simplificadas de la igualdad para $x = 0$ (pruebe $z \vee yz = z$) y $x = 1$ (pruebe $y \vee yz = y$). Es posible que más adelante tenga que dividir un problema más complejo.
- d) Use manipulación lógica para convertir una expresión en la otra: $(x \vee y)(y \vee z)(z \vee x) = (xy \vee xz \vee yy \vee yz)(z \vee x) = (xz \vee y)(z \vee x) = xzz \vee xzx \vee yz \vee yx = xz \vee yz \vee yx$.

1.3 Diseño de redes de compuertas

Cualquier expresión lógica compuesta de NOT, AND, OR, XOR y otros tipos de compuertas representa una especificación para una red de compuertas. Por ejemplo, $xy \vee yz \vee zx$ especifica la red de

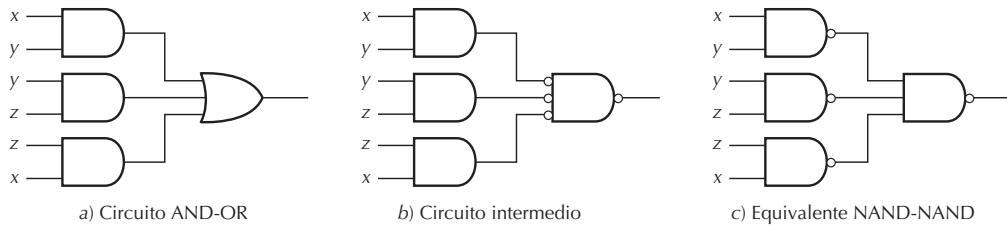


Figura 1.6 Circuito AND-OR de dos niveles y dos circuitos equivalentes.

compuertas de la figura 1.6a. Éste es un circuito lógico AND-OR de dos niveles con compuertas AND en el nivel 1 y una compuerta OR en el nivel 2. Puesto que, de acuerdo con la ley de DeMorgan (tabla 1.2, último renglón, columna media), una compuerta OR se puede sustituir por compuertas NAND con entradas complementadas, se observa que la figura 1.6b equivale a la figura 1.6a. Ahora, al mover las burbujas de inversión en las entradas de la compuerta NAND del nivel 2 en la figura 1.6b, a las salidas de las compuertas AND en el nivel 1, se deriva el circuito NAND-NAND de nivel dos de la figura 1.6c que realiza la misma función. Un proceso similar convierte cualquier circuito OR-AND de nivel dos a un circuito equivalente NOR-NOR. En ambos casos, se debe invertir cualquier señal que entre directamente a una compuerta de nivel 2 (porque la burbuja permanece).

Mientras que el proceso de convertir una expresión lógica en un diagrama lógico, y, por tanto, una realización de hardware asociada, es trivial, no lo es el hecho de obtener una expresión lógica que conduzca al mejor circuito hardware posible. Por un lado, la definición “mejor” cambia según la tecnología y el esquema de implementación que se use (por ejemplo, VLSI a la medida, lógica programable, compuertas discretas) y en las metas de diseño (por ejemplo, alta rapidez, ahorro de potencia, bajo costo). Asimismo, si el proceso de simplificación no se hace mediante herramientas de diseño automáticas, no sólo resultará engorroso sino también imperfecto; por ejemplo, se puede basar en minimizar el número de compuertas empleadas, sin tomar en consideración las implicaciones de rapidez y costo de los alambres que conectan las compuertas. Este libro no se preocupará por el proceso de simplificación para las expresiones lógicas. Lo anterior se sustenta en que cada función lógica que encontrará, cuando se divida de manera adecuada, es lo suficientemente simple para permitir que las partes requeridas se realicen mediante circuitos lógicos eficientes en una forma directa. El proceso se ilustra con dos ejemplos.

Ejemplo 1.2: Decodificador BCD a siete segmentos La figura 1.7 muestra cómo los dígitos decimales 0-9 pueden aparecer en un display de siete segmentos. Diseñe los circuitos lógicos para generar las señales de habilitación que provoquen que los segmentos se enciendan o apaguen, de acuerdo con una representación binaria de cuatro bits del dígito decimal (decimal codificado en binario o código BCD) a desplegar como entrada.



Figura 1.7 Display de siete segmentos de dígitos decimales. Los tres segmentos abiertos se pueden usar de manera opcional. El dígito 1 se puede mostrar en dos formas; aquí se muestra la versión más común del lado derecho.

Solución: La figura 1.7 constituye una representación gráfica de los renglones 0-9 de una tabla de verdad de 16 renglones, donde cada uno de los renglones 10-15 significa una condición del tipo *no importa*. Existen cuatro columnas de entrada x_3, x_2, x_1, x_0 y siete columnas de salida $e_0 - e_6$. La figura 1.8 muestra

la numeración de los segmentos y el circuito lógico que produce la señal de habilitación para el segmento número 3. La columna de salida de la tabla de verdad asociada con e_3 contiene las entradas 1, 0, 1, 1, 0, 1, 1, 0, x, x, x, x, x (el 9 se muestra sin el segmento número 3). lo anterior representa la expresión lógica $e_3 = x_1x'_0 \vee x'_2x'_0 \vee x'_2x_1 \vee x_2x'_1x_0$. Observe que e_3 es independiente de x_3 . Derivar los circuitos lógicos para los restantes seis segmentos se hace de manera similar.

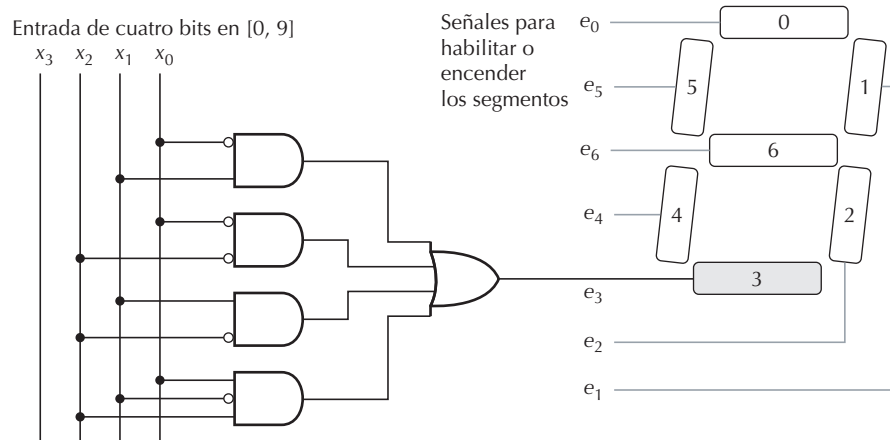


Figura 1.8 Circuito lógico que genera la señal de habilitación para el segmento inferior (número 3) en una unidad de display de siete segmentos.

Ejemplo 1.3: Actuador de una simple máquina expendedora Una pequeña máquina expendedora dispensa un paquete de chicle o una barra de dulce, cuyo costo individual es de 35 centavos. El cliente debe depositar el costo exacto, éste consiste en una moneda de 25 centavos y otra de 10 centavos; cuando esto último se realiza, entonces se indicará la preferencia por uno de los dos artículos al presionar al botón correspondiente, luego se jalará una palanca para liberar el artículo deseado y éste caerá en un recipiente. El actuador representa un circuito combinacional con tres salidas: r_{chicle} (r_{barra}), cuando se postula libera un paquete de chicle (barra de dulce); el hecho de postular r_{monedas} propicia que las monedas depositadas se regresen cuando, por cualquier razón, una venta no se concluye. Las entradas al actuador son las señales siguientes:

- s_{palanca} indica el estado de la palanca (1 significa que la palanca se jaló)
- c_{25} y c_{10} , para las monedas, proporcionada por un módulo de detección de moneda
- a_{chicle} y a_{barra} proporcionada por dispositivos que perciben la disponibilidad de los dos artículos
- p_{chicle} y p_{barra} que provienen de dos botones que expresan la preferencia del cliente

Solución: Remítase de nuevo a la tabla 1.1, que constituye la tabla de verdad para el actuador de la máquina expendedora. Cuando la palanca no se jale, todas las salidas deben ser 0, sin importar los valores de otras entradas (línea 1). El resto de los casos que siguen corresponden a $s_{\text{palanca}} = 1$. Cuando no se depositan monedas, ningún artículo se debe liberar; el valor de r_{monedas} es inmaterial en este caso, pues no hay moneda para regresar (línea 2). Cuando sólo se ha depositado una moneda, no se debe liberar artículo alguno y la moneda se debe regresar al cliente (líneas 3-4). El resto de los casos corresponden al depósito de 35 centavos y jalar la palanca. Si el cliente no ha seleccionado, o si seleccionó ambos

artículos, las monedas se deben regresar y no se debe liberar artículo alguno (líneas 5-6). Si se seleccionó una barra de dulce, se libera una barra de dulce o se regresan las monedas, dependiendo de a_{barra} (líneas 7-8). El caso para la selección de un paquete de chicle es similar (líneas 9-10). Las siguientes expresiones lógicas para las tres salidas se obtienen fácilmente por inspección: $r_{\text{chicle}} = s_{\text{palanca}} c_{25} c_{10} p_{\text{chicle}}$, $r_{\text{barra}} = s_{\text{palanca}} c_{25} c_{10} p_{\text{barra}}$, $r_{\text{monedas}} = s_{\text{palanca}} (c'_{25} \vee c'_{10} \vee p'_{\text{chicle}} p'_{\text{barra}} \vee p_{\text{chicle}} p_{\text{barra}} \vee a'_{\text{chicle}} p_{\text{chicle}} \vee a'_{\text{barra}} p_{\text{barra}})$.

1.4 Partes combinacionales útiles

Ciertas partes combinacionales se pueden usar en la síntesis de los circuitos digitales, en gran medida como cuando se utilizan en casa los armarios y aditamentos prefabricados para baño. Tales bloques constructores del tipo estándar son numerosos e incluyen muchos circuitos aritméticos que se analizarán en la parte 3 del libro. En esta sección se revisa el diseño de tres tipos de componentes combinacionales que se usan con propósitos de control: multiplexores, decodificadores y codificadores.

Un multiplexor 2^a a 1, mux para abreviar, tiene 2^a entradas de datos, x_0, x_1, x_2, \dots , una sola salida z , y a señales de selección o dirección y_{a-1}, \dots, y_1, y_0 . La salida z es igual a la entrada x_i cuyo índice i tiene la representación binaria $(y_{a-1} \dots y_1 y_0)_{\text{dos}}$. Los ejemplos incluyen multiplexores 2 a 1 (dos vías) y 4 a 1 (cuatro vías), que se muestran en la figura 1.9, que tienen una y dos entradas de dirección, respectivamente. Al igual que los arreglos de compuertas, muchos mux controlados por las mismas líneas de dirección son útiles para seleccionar un haz de señales sobre otro (figura 1.9d). Un mux n a 1, donde n no es una potencia de 2, se puede construir al recortar las partes innecesarias de un mux más grande con 2^a entradas, donde $2^{a-1} < n < 2^a$. Por ejemplo, el diseño de la figura 1.9f se puede convertir en un mux de tres entradas al remover el mux con entradas x_2 y x_3 , y luego conectar x_2 directamente con el mux de segundo nivel. En cualquier momento la salida z de un mux es igual a una de sus entradas.

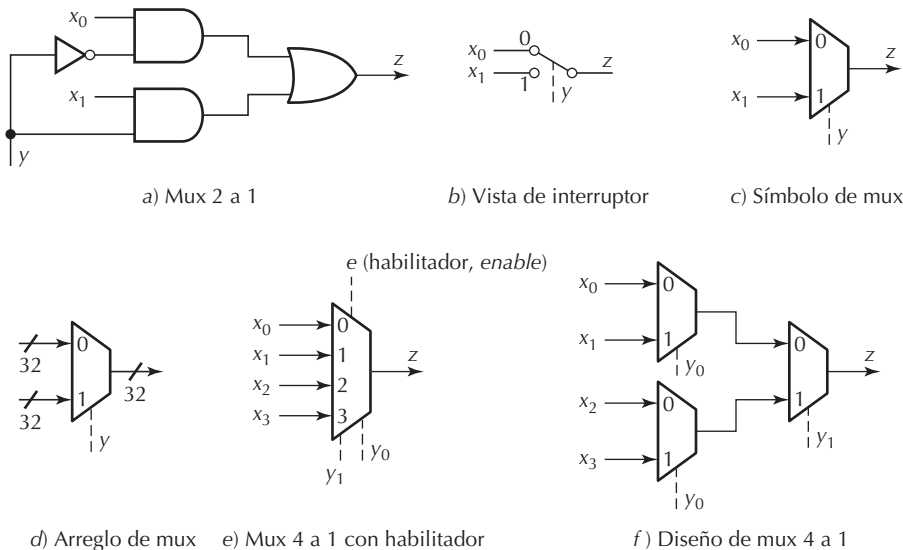


Figura 1.9 Un multiplexor, o selector, selecciona una de muchas entradas y la dirige hacia la salida, según el valor binario de un conjunto de señales de selección o dirección que se le proporcionen.

Al proporcionar al multiplexor una señal de habilitación e , que se suministra como entrada adicional a cada una de las compuertas AND de la figura 1.9a, se obtiene la opción de forzar la salida a 0, independientemente de las entradas de datos y direcciones. En esencia, esto último equivale a que ninguna de las entradas se seleccione (figura 1.9e).

Los multiplexores representan bloques constructores versátiles. Cualquier función booleana de a variables se puede implementar mediante mux 2^a a 1, donde las variables se conectan a las entradas de dirección y cada una de las 2^a entradas de datos lleva un valor constante 0 o 1 de acuerdo con el valor de la tabla de verdad de la función para dicho renglón particular. De hecho, si el complemento de una de las variables está disponible en la entrada, entonces un mux más pequeño 2^{a-1} a 1 es suficiente. Como ejemplo concreto, para implementar la función e_3 definida en el ejemplo 1.2, es posible usar un mux 8 a 1 con entradas de dirección conectadas a x_2, x_1, x_0 y entradas de datos que llevan 1, 0, 1, 1, 0, 1, 1, 0, de arriba abajo. De manera alterna, se puede usar un mux 4 a 1, con líneas de dirección conectadas a x_1 y x_0 y líneas de datos que portan $x'_2, x_2, 1$ y x'_2 , nuevamente de arriba abajo. Los últimos cuatro términos se derivan fácilmente de la expresión para e_3 mediante la fijación exitosa del valor de x_1x_0 en 00, 01, 10 y 11.

Un decodificador a a 2^a (a entradas) postula una y sólo una de sus 2^a líneas de salida. La salida x_i que se postula tiene un índice i cuya representación binaria empata el valor en las a líneas de dirección. En la figura 1.10a se muestra el diagrama lógico para un decodificador 2 a 4, y en la figura 1.10b se proporciona su símbolo abreviado. Si las salidas de tal decodificador se usan como señales de habilitación para cuatro diferentes elementos o unidades, entonces el decodificador permite elegir cuál de las cuatro unidades se habilita en un momento específico. Si se quiere tener la opción de no habilitar alguna de las cuatro unidades, entonces se puede usar un decodificador con una entrada de habilitación (figura 1.10c), también conocido como *demultiplexor* o *demux*, donde la entrada de habilitación e se proporciona como una entrada adicional a cada una de las cuatro compuertas AND de la figura 1.10a (más generalmente 2^a compuertas AND). La denominación demultiplexor indica que este circuito realiza la función opuesta de un mux: mientras que un mux selecciona una de sus entradas y la dirige hacia la salida, un demux recibe una entrada e y la orienta a una salida seleccionada.

La función de un *codificador* es exactamente la opuesta de un decodificador. Cuando se postula una, y sólo una, entrada de un codificador de 2^a entradas (2^a a a), su salida de bit a proporciona el índice de

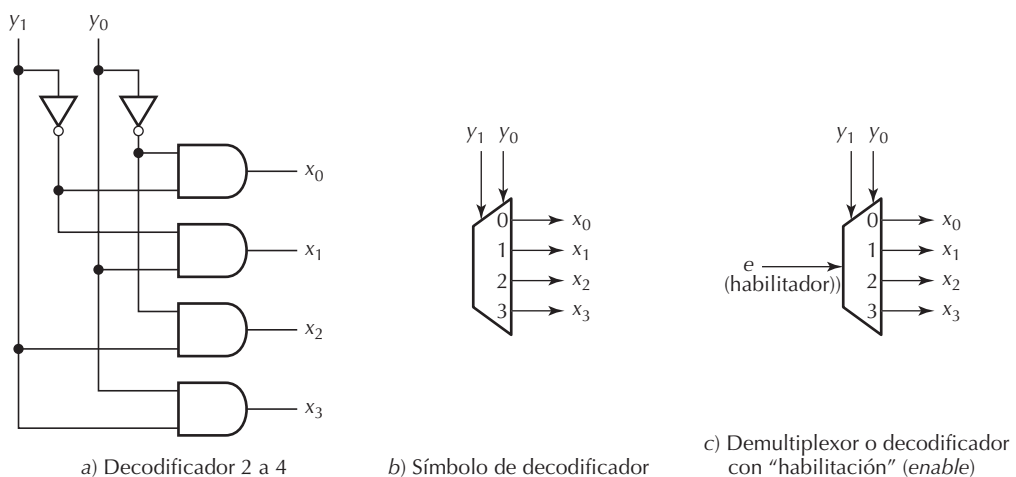


Figura 1.10 Un decodificador permite la selección de una de 2^a opciones que usan una dirección de bit a como entrada. Un demultiplexor (demux) constituye un decodificador que sólo selecciona una salida si se postula su señal de habilitación.

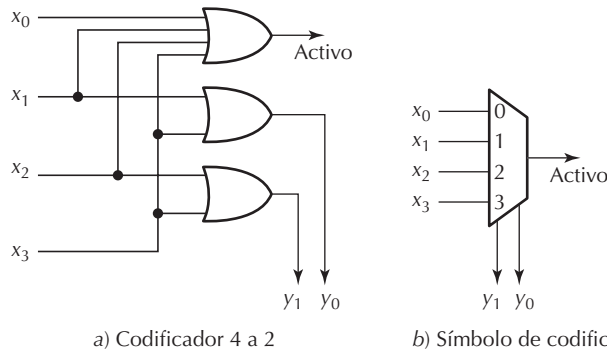


Figura 1.11 Un codificador 2^a a a da salida a un número binario de a bits igual al índice del único 1 entre sus 2^a entradas.

la entrada postulada en la forma de un número binario. En la figura 1.11a se muestra el diagrama lógico de un codificador 4 a 2, y en la figura 1.11b su símbolo abreviado. De manera más general, el número n de entradas no necesita ser una potencia de 2. En este caso, la salida del codificador de $\lceil \log_2 n \rceil$ bits constituye la representación binaria del índice de la única entrada postulada; esto es, un número entre 0 y $n - 1$. Si se diseña un codificador como una colección de compuertas OR, como en la figura 1.11a, produce la salida todos 0 cuando ninguna entrada se postula o cuando la entrada 0 se postula. Estos dos casos son, por ende, indistinguibles en la salida del codificador. Si se levanta la restricción de que cuando mucho una entrada del codificador se puede postular y se diseña el circuito para dar salida al índice de la entrada postulada con el índice más bajo, entonces resulta un *codificador de prioridad*. Por ejemplo, si supone que se postulan las entradas x_1 y x_2 , el codificador de la figura 1.11a produce la salida 11 (índice = 3, que no corresponde a alguna entrada postulada), mientras que un codificador de prioridad daría salida 01 (índice = 1, el más bajo de los índices para entradas postuladas). La señal “activo” diferencia entre los casos de ninguna de las entradas postuladas y x_0 postulada.

Tanto decodificadores como codificadores son casos especiales de *convertidores de código*. Un decodificador convierte un código binario de a bits en un código de 1 de 2^a , un código con 2^a palabras código cada una de las cuales se compone de un solo 1 y todos los otros bits 0. Un codificador convierte un código 1 de 2^a a un código binario. En el ejemplo 1.2 se diseña un convertidor de código BCD a siete segmentos.

■ 1.5 Partes combinacionales programables

Para evitar el uso de gran número de circuitos integrados de pequeña escala con el propósito de implementar una función booleana de muchas variables, los fabricantes de CI (circuitos integrados) ofrecen muchos arreglos de compuertas cuyas conexiones se pueden adecuar mediante la programación. Con respecto al mecanismo de programación, existen dos tipos de circuitos. En uno, todas las conexiones de interés potencial ya están hechas, pero se pueden remover de manera selectiva. Tales conexiones se hacen mediante *fusibles* que se pueden quemar al pasar una corriente suficientemente grande a través de ellos. En otro, los elementos *antifusible* se usan para establecer selectivamente conexiones donde se desee. En los diagramas lógicos, la misma convención se usa para ambos tipos: una conexión que se queda en lugar, o se establece, aparece como un punto grueso sobre líneas que se cruzan, mientras que para una conexión que se quema, o no se establece, no hay tal punto. La figura 1.12a muestra cómo se pueden implementar las funciones $w \vee x \vee y$ y $x \vee z$ mediante compuertas OR programables. Un arreglo de éstas, conectado a las salidas de un decodificador de a a 2^a , permite implementar muchas funciones de a variables de entrada a la vez (figura 1.12c). Este ordenamiento se conoce como memoria de sólo lectura programable o PROM.

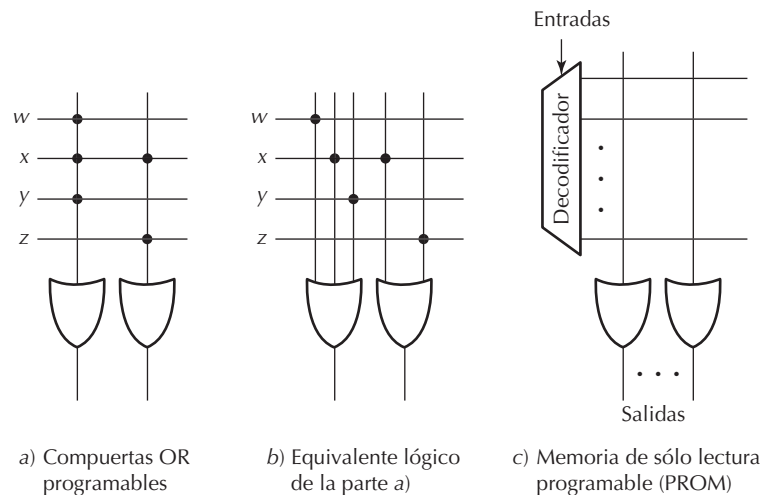


Figura 1.12 Conexiones programables y su uso en una PROM.

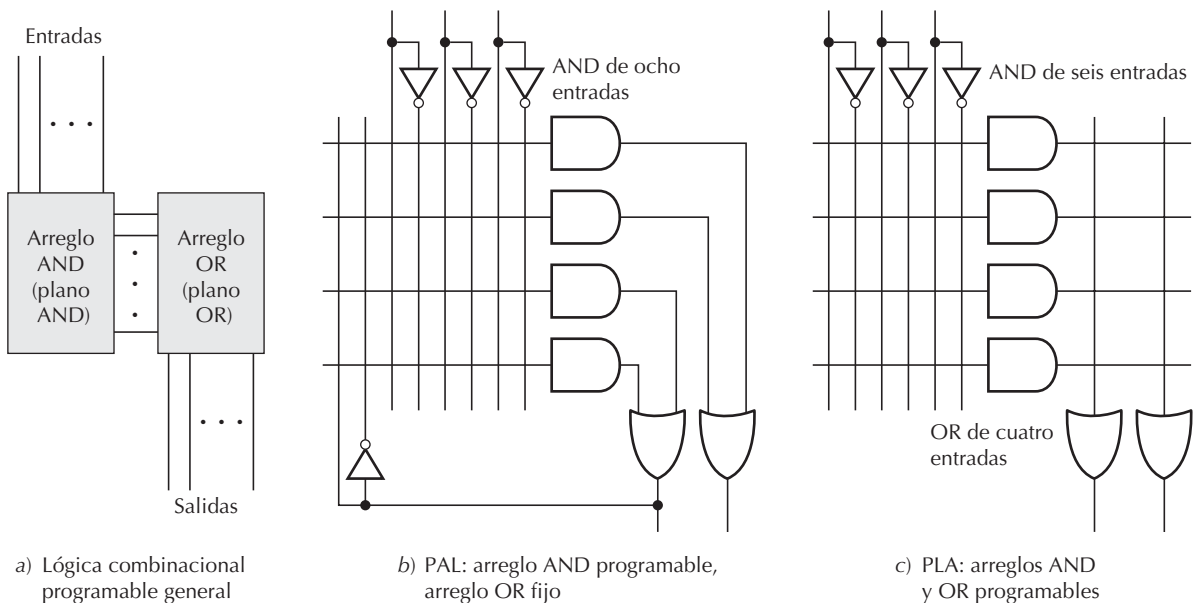


Figura 1.13 Lógica combinacional programable: estructura general y dos clases conocidas como dispositivos PAL y PLA. No se muestran la PROM con arreglo AND fijo (un decodificador) y arreglo OR programable.

La figura 1.13a muestra una estructura más general para circuitos de lógica combinacional programable en la que el decodificador de la figura 1.12c se sustituyó por un arreglo de compuertas AND. Las n entradas, y sus complementos formados internamente, se proporcionan al arreglo AND, que genera un número de términos producto que involucran variables y sus complementos. Estos términos producto constituyen entrada a un arreglo OR que combina los términos producto adecuados para cada una de hasta m funciones de interés que sean salida. PROM representa un caso especial de esta estructura donde el

arreglo AND es un decodificador fijo y el arreglo OR se puede programar de manera arbitraria. Cuando el arreglo OR tiene conexiones fijas pero las entradas a las compuertas AND se pueden programar, el resultado es un dispositivo de lógica de arreglo programable o PAL (figura 1.13b). Cuando tanto los arreglos AND como OR se pueden programar, el circuito resultante se denomina arreglo lógico programable o PLA (figura 1.13c). Los dispositivos PAL y PLA son más eficientes que las PROM porque generan menos términos producto. Los primeros son más eficientes, pero menos flexibles, que los segundos.

En las partes PAL comerciales, debido al limitado número de términos producto que se pueden combinar para formar cada salida, con frecuencia se proporciona un mecanismo de retroalimentación que hace que algunas de las salidas se seleccionen como entradas hacia compuertas AND. En la figura 1.13b la salida izquierda se retroalimenta hacia el arreglo AND, donde se puede usar como entrada en las compuertas AND que contribuyen a la formación de la salida derecha. De manera alterna, tales salidas retroalimentadas se pueden usar como entradas primarias para el caso de que se necesitaran entradas adicionales. Un producto PAL usado de manera común es el dispositivo PAL16L8. Los números 16 y 8 en el nombre del dispositivo se refieren a líneas de entrada y salida, respectivamente; el paquete tiene 20 pines para diez entradas, dos salidas, seis líneas I/O bidireccionales, potencia y tierra. El arreglo AND programable de este dispositivo consiste de 64 compuertas AND, cada una con 32 entradas (las 16 entradas y sus complementos). Las 64 compuertas AND se dividen en ocho grupos de ocho compuertas. Dentro de cada grupo, siete compuertas AND alimentan una compuerta OR de siete entradas que produce una salida, y el resto de la compuerta AND genera una señal de habilitación para un buffer tres estados inversor.

Los PLA no se usan como partes de artículos sino como estructuras que permiten la implementación regular y sistemática de funciones lógicas en chips VLSI (*very large scale integration*, integración a muy grande escala) a la medida. Por ejemplo, más tarde se verá (en el capítulo 13) que la instrucción decodificación lógica de un procesador es un candidato natural para la implementación PLA.

■ 1.6 Estimaciones de temporización y de circuito

Cuando varían las señales de entrada a una compuerta, cualquier cambio de salida requerida no ocurre inmediatamente sino que más bien toma efecto con cierto retardo. El retardo de compuerta varía con la tecnología subyacente, tipo de compuerta, número de entradas (*carga de entrada* de compuerta), voltaje suministrado, temperatura de operación, etc. Sin embargo, como aproximación de primer orden, todos los retardos de compuerta se pueden considerar iguales y se denotan mediante el símbolo δ . De acuerdo con lo anterior, un circuito lógico de nivel dos tiene un retardo de 2δ . Para la tecnología CMOS (*complementary metal-oxide semiconductor*, semiconductor de óxido metálico complementario) que se usa en la mayoría de los circuitos digitales modernos, el *retardo de compuerta* puede ser tan pequeño como una fracción de nanosegundo. La propagación de señal en los alambres que conectan las compuertas también aporta cierto retardo pero, de nuevo en el contexto de un análisis aproximado, tales retardos se pueden ignorar para simplificar los análisis; sin embargo, conforme las dimensiones del circuito se reducen, tal omisión se vuelve más problemática. La única forma precisa de estimar el retardo de un circuito lógico es correr el diseño completo, con todos los detalles de los elementos lógicos y alambrados, mediante una herramienta de diseño. Entonces, se deben incluir márgenes de seguridad en las estimaciones de temporización para explicar las irregularidades de proceso y otras variaciones.

Cuando los retardos a lo largo de las trayectorias de un circuito lógico son desiguales, sea que se deban al diferente número de compuertas a través de las que pasan las diferentes señales o las variaciones antes mencionadas, ocurre un fenómeno conocido como *interferencia* (*glitching*). Suponga que debe implementar la función $f = x \vee y \vee z$ con el circuito de la figura 1.13b. Puesto que las compuertas OR en el circuito blanco sólo tienen dos entradas, primero debe generar $a = x \vee y$ en la salida izquierda y luego usar el resultado para formar $f = a \vee z$ en la salida derecha. Note que las señales x y y pasan a través de

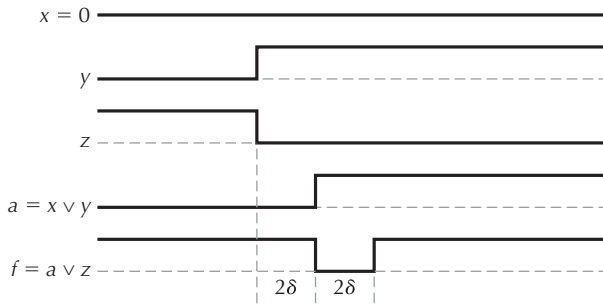


Figura 1.14 Diagrama de temporización para un circuito que muestra interferencia.

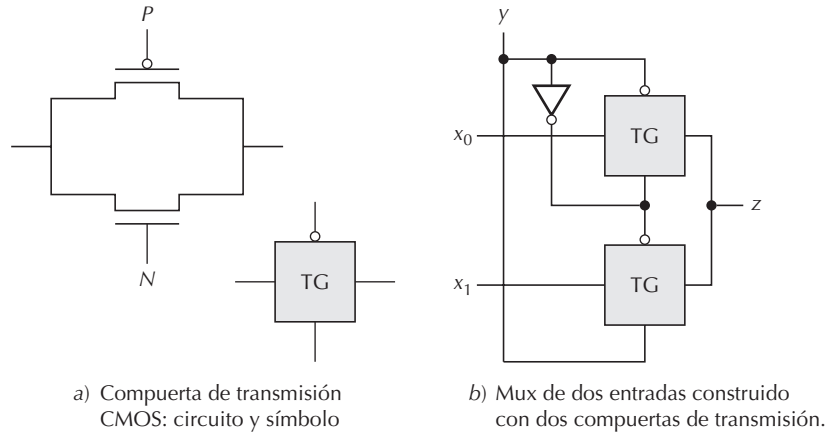


Figura 1.15 Compuerta de transmisión CMOS y su uso para construir un mux 2 a 1.

cuatro niveles de compuertas, mientras que z pasa a través de sólo dos niveles. El proceso anterior conduce a la situación que se muestra en el diagrama de temporización de la figura 1.14, donde $x = 0$ en toda la extensión, mientras que y cambia de 0 a 1 en casi el mismo tiempo que z cambia de 1 a 0. Teóricamente, la salida debe ser 1 en todo momento. Sin embargo, debido a los diferentes retardos, la salida supone el valor 0 para 2δ unidades de tiempo. Ésta es una razón por la que se necesitan análisis de temporización precisos y márgenes de seguridad para garantizar que se permite un tiempo adecuado para que los cambios se propaguen por completo a través del circuito, y con el propósito de que las salidas supongan sus valores finales correctos, antes de que los resultados generados se usen en otros cálculos.

Aun cuando en este libro se trate exclusivamente con circuitos lógicos combinacionales contruïdos de compuertas, se debe mencionar, para complementar, que con la tecnología CMOS se pueden derivar y usar otros elementos de circuito que no corresponden directamente con una compuerta o red de compuertas. En la figura 1.15 se presentan dos ejemplos. El circuito de dos transistores bosquejado en la figura 1.15a, junto con su representación simbólica, se denomina *compuerta de transmisión* (TG, por sus siglas en inglés). Dicha compuerta conecta sus dos lados cuando la señal de control N se postula y los desconecta cuando P se postula. Para el caso de que las señales N y P sean complementarias, la compuerta de transmisión se comporta como un interruptor controlado. Se pueden usar dos compuertas de transmisión y un inversor (otro circuito CMOS de dos transistores) para formar un mux de 2 a 1, como se muestra en la figura 1.15b. Un mux de 2^a entradas se puede construir con el uso de 2^a compuertas de transmisión y un decodificador que convierte la selección de número binario $(y_{a-1} \cdots y_1 y_0)_{\text{dos}}$ en una sola señal postulada que alimenta una de las compuertas de transmisión.

PROBLEMAS

1.1 Elementos lógicos universales

Los tres elementos lógicos AND, OR y NOT forman un conjunto universal porque cualquier función lógica se puede implementar mediante estos elementos, y no se requiere nada más.

- Demuestre que remover AND u OR del conjunto deja universal el resto del conjunto de dos elementos.
- Demuestre que XOR forma un conjunto universal o con AND o con OR.
- Demuestre que la compuerta NAND es universal.
- Demuestre que la compuerta NOR es universal.
- Demuestre que el multiplexor 2 a 1 es universal.
- ¿Existe algún otro elemento de dos entradas, además de NAND y NOR, que sea universal? *Sugerencia:* Existen diez funciones de dos variables que depende de ambas variables.

1.2 Actuador de máquina expendedora

Extienda el diseño del actuador de máquina expendedora del ejemplo 1.3 en las siguientes formas:

- Cada uno de los dos artículos cuesta 65 centavos.
- Hay cuatro artículos a elegir.
- Los dos cambios de las partes a) y b).
- Modifique el diseño de la parte a) para aceptar cualquier cantidad, hasta un dólar (incluidos billetes), y para regresar cambio.

1.3 Realización de funciones booleanas

Diseñe un circuito lógico de tres entradas y una salida que implemente alguna función booleana deseada de las tres entradas con base en un conjunto de entradas de control. *Sugerencia:* Existen 256 diferentes funciones booleanas de tres variables, de modo que la entrada de control al circuito (el *opcode*, código de operación) debe contener al menos ocho bits. La tabla de verdad de una función booleana de tres variables tiene ocho entradas.

1.4 Decodificador BCD a siete segmentos

Considere el decodificador BCD a siete segmentos parcialmente diseñado en el ejemplo 1.2.

- Complete el diseño y suponga que no se incluye ninguno de los tres segmentos abiertos de la figura 1.7.

- Rehaga el diseño con un decodificador de cuatro a 16 y siete compuertas OR.
- Determine qué se desplegará cuando los circuitos lógicos derivados en la parte a) se alimenten con cada una de las seis entradas prohibidas de la 1010 a la 1111.
- Repita la parte c) para el diseño de la parte b).

1.5 Decodificador BCD a siete segmentos

Diseñe formas modificadas del decodificador BCD a siete segmentos del ejemplo 1.2 bajo cada uno de los siguientes conjuntos de suposiciones. En todo caso suponga que “1” se representa mediante los dos segmentos en el extremo derecho del panel.

- Las entradas representan dígitos BCD; los dígitos 6, 7 y 9 se despliegan con 6, 3 y 6 segmentos, respectivamente (vea la figura 1.7).
- Las entradas constituyen dígitos hexadecimales, con los dígitos agregados 10-15 representados con letras mayúsculas (A, C, E, F) o minúsculas (b, d). Use las suposiciones de la parte a) para los dígitos 6, 7 y 9.
- Las entradas son dígitos BCD y las formas de display se eligen de tal modo que cualquier señal de habilitación de segmento individual que queda permanentemente pegada en 0 no conduce a un error de display indetectable.
- Repita la parte c) para una señal de habilitación individual que se queda permanentemente pegada en 1.

1.6 Generación de paridad y verificación

Un *generador de paridad* par de n entradas produce una salida 1 si y sólo si (ssi) un número impar de sus entradas es 1. El circuito se llama así porque al unir la salida producida a la entrada de n bit produce una palabra de $(n + 1)$ bits de paridad par. Un *verificador de paridad* par de n entradas produce una salida 1 (señal de error) ssi un número impar de sus entradas es 1.

- Diseñe un generador de paridad par de ocho entradas sólo con el uso de compuertas XOR.
- Repita la parte a) para un generador de paridad impar.
- Diseñe un verificador de paridad par de nueve entradas sólo con el uso de compuertas XOR. ¿Cómo se relaciona su diseño con los de las partes a) y b)?
- Repita la parte c) para un verificador de paridad impar.
- Demuestre que los circuitos de generador de paridad más rápidos se pueden construir de forma recursiva,

con base en el diseño de un circuito de $(n/2)$ entradas de tipos adecuados. ¿Qué ocurre cuando n es impar?

1.7 Multiplicidad de funciones booleanas

Existen cuatro funciones booleanas de una sola variable, de las cuales sólo dos dependen realmente de la variable de entrada x (x y x'). De igual modo, existen 16 funciones de dos variables, de las cuales sólo diez dependen de ambas variables (excluidas x , x' , y , y' , 0 y 1).

- ¿Cuántas funciones booleanas de tres variables existen, y cuántas de ellas en realidad dependen de las tres variables?
- Generalice el resultado de la parte a) para funciones de n variables. *Sugerencia:* Debe sustraer del número de funciones de n variables todas aquellas que dependan de $n-1$ o menos variables.

1.8 Expresiones lógicas equivalentes

Pruebe cada una de las cuatro equivalencias del ejemplo 1.1 con el uso de los otros tres métodos mencionados en el ejemplo.

1.9 Expresiones lógicas equivalentes

Use el método de sustitución aritmética para probar los siguientes pares de equivalentes de expresiones lógicas.

- $xy' \vee x'z' \vee y'z' \equiv xy' \vee x'z'$
- $xyz \vee x' \vee y' \vee z' \equiv 1$
- $x \oplus y \oplus z \equiv xyz \vee xy'z' \vee x'yz' \vee x'y'z$
- $xz \vee wy'z' \vee wxy' \vee w'xy \vee x'yz' \equiv xz \vee wy'z' \vee w'yz' \vee wx'z'$

1.10 Expresiones lógicas equivalentes

Pruebe que los siguientes pares de expresiones lógicas son equivalentes, primero mediante el método de tabla de verdad y luego mediante la manipulación simbólica (use las leyes del álgebra booleana).

- $xy' \vee x'z' \vee y'z' \equiv xy' \vee x'z'$
- $xyz \vee x' \vee y' \vee z' \equiv 1$
- $x \oplus y \oplus z \equiv xyz \vee xy'z' \vee x'yz' \vee x'y'z$
- $xz \vee wy'z' \vee wxy' \vee w'xy \vee x'yz' \equiv xz \vee wy'z' \vee w'yz' \vee wx'z'$

1.11 Diseño de un interruptor 2×2

Diseñe un circuito combinacional que actúe como un interruptor 2×2 . El interruptor tiene entradas de datos a y

b , una señal de control “cruzado” c y salidas de datos x y y . Cuando $c = 0$, a se conecta a x y b a y . Cuando $c = 1$, a se conecta a y y b a x (es decir, las entradas se cruzan).

1.12 Circuitos de comparación numérica

Suponga $x = (x_2x_1x_0)_{\text{dos}}$ y $y = (y_2y_1y_0)_{\text{dos}}$ son números binarios de tres bits sin signo. Escriba una expresión lógica en términos de las seis variables booleanas x_2 , x_1 , x_0 , y_2 , y_1 , y_0 que suponga el valor 1 ssi:

- $x = y$
- $x < y$
- $x \leq y$
- $x - y \leq 1$
 $x - y$ es constante
 $x + y$ es divisible entre 3

1.13 Circuitos de comparación numérica

Repita el problema 1.2 (todas las partes), pero suponga que las entradas de tres bits x y y son números en complemento a 2 en el rango -4 a $+3$.

1.14 Suma de productos y producto de sumas

Expresé cada una de las siguientes expresiones lógicas en formas de suma de productos y producto de sumas.

- $(x \vee y')(y \vee wz)$
- $(xy \vee z)(y \vee wz')$
- $x(x \vee y')(y \vee z') \vee x'$
- $x \oplus y \oplus z$

1.15 Código Hamming SEC/DED

Un tipo particular de código Hamming tiene palabras código de ocho bits $P_8D_7D_6D_5P_4D_3P_2P_1$ que codifican 16 diferentes valores de datos. Los bits de paridad P_i se obtienen de los bits de datos D_j de acuerdo con las ecuaciones lógicas $P_1 = D_3 \oplus D_5 \oplus D_6$, $P_2 = D_3 \oplus D_5 \oplus D_7$, $P_4 = D_3 \oplus D_6 \oplus D_7$, $P_8 = D_5 \oplus D_6 \oplus D_7$.

- Demuestre que este código es capaz de corregir cualquier error de bit sencillo (SEC, *Single Error Corrector*) y derive las reglas de corrección. *Sugerencia:* Piense en términos de calcular cuatro resultados de verificación de paridad, como $C_1 = P_1 \oplus D_3 \oplus D_5 \oplus D_6$, que deben producir 0 para una palabra código libre de error.
- Demuestre que el código detecta todos los errores de doble bit (DED, *Double Error Detector*) además

de corregir los errores sencillos (por ende, es un código SEC/DED).

- c) Diseñe el circuito de codificación, sólo con el uso de compuertas NAND de dos entradas.
- d) Diseñe el circuito decodificador que incluya corrección de error sencillo y postulación de una señal indicadora de error en caso de dobles errores.
- e) Derive una implementación PROM del circuito decodificador de la parte d).

1.16 Implementación de funciones lógicas con base en mux

Muestre las entradas requeridas para implementar las dos siguientes funciones lógicas con el uso de multiplexores 4 a 1 si sólo la entrada x está disponible en forma de complemento.

- a) $f(x, y, z) = y'z \vee x'y \vee xz'$
- b) $g(w, x, y, z) = wx'z \vee w'yz \vee xz'$

1.17 Implementación de funciones lógicas con base en mux

- a) Demuestre que cualquier función lógica de tres variables $f(x, y, z)$ se puede realizar con el uso de tres multiplexores de dos entradas, si se supone que x' está disponible como entrada.
- b) ¿Bajo qué condiciones se puede realizar una función de tres variables con el uso de tres multiplexores de dos entradas, sin requerir entrada complementada alguna?
- c) Proporcione un ejemplo de una función de tres variables (que verdaderamente dependa de las tres variables) que se pueda realizar sólo con el uso de dos multiplexores de dos entradas, sin entrada complementada disponible. También muestre un diagrama de la realización de la función con dos multiplexores.
- d) ¿Una función de tres variables que es realizable con un solo multiplexor de dos entradas puede depender verdaderamente de las tres variables? Proporcione un ejemplo o demuestre por qué tal arreglo es imposible.

1.18 Comparador numérico iterativo

Un comparador iterativo para números binarios no signados de k bits consiste de k celdas ordenadas en un circuito lineal o en cascada. Una celda recibe un bit de cada uno de los dos operandos x y y , y un par de señales G_E ($x \geq y$ hasta aquí) y L_E ($x \leq y$ hasta aquí) de una celda vecina, y produce señales G_E y L_E para la otra vecina. Note que $G_E = L_E = 1$ significa que los dos números son iguales hasta aquí.

- a) Diseñe la celda requerida y suponga que las señales G_E y L_E se propagan de derecha a izquierda.
- b) Repita la parte a) para propagar la señal de izquierda a derecha.
- c) Demuestre que las celdas de la parte a) o b) se pueden conectar en una estructura de árbol, en oposición al arreglo lineal, para producir un comparador más rápido.

1.19 Expresiones aritméticas para compuertas lógicas

Las expresiones aritméticas que caracterizan a las compuertas lógicas (figura 1.1) se pueden extender a compuertas con más de dos entradas. Esto es trivial para las compuertas AND. Escriba las expresiones aritméticas equivalentes para compuertas OR de tres y cuatro entradas. Generalice la expresión para una compuerta OR de h entradas.

1.20 Partes combinacionales programables

- a) Muestre cómo realizar la función lógica $f(x, y, z) = y'z \vee x'y \vee xz'$ en el PLA de la figura 1.13c.
- b) Repita la parte a) en la PAL de la figura 1.13b.
- c) Demuestre que cualquier función que se pueda realizar por la PAL de la figura 1.13b también es realizable por el PLA de la figura 1.13c. Observe que esto último no es una afirmación acerca de las PAL y los PLA en general, sino más bien acerca de las instancias específicas que se muestran en la figura 1.13.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Brow00] Brown, S. y Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill, 2000.
- [Erce99] Ercegovac, M. D., T. Lang y J. H. Moreno, *Introduction to Digital Systems*, Wiley, 1999.
- [Haye93] Hayes, J. P., *Introduction to Digital Logic Design*, Addison-Wesley, 1993.

- [Katz94] Katz, R. H., *Contemporary Logic Design*, Benjamin/Cummings, 1994.
- [Parh99] Parhami, B. y D. M. Kwai, "Combinational Circuits", en *Encyclopedia of Electrical and Electronics Engineering*, Wiley, vol. 3 (Ca-Co), 1999, pp. 562-569.
- [Wake01] Wakerly, J. F., *Digital Design: Principles and Practices*, Prentice Hall, updated 3a. ed., 2001.
- [WWW] Sitios Web de algunos fabricantes de PAL y PLA: altera.com, atmel.com, cypress.com, latticesemi.com, philips.com, vantis.com.

CIRCUITOS DIGITALES CON MEMORIA

“Los microprocesadores no fueron un producto de la industria de la computación. Son resultado del deseo, y la necesidad imperiosa, de la nueva industria de semiconductores por encontrar una aplicación rentable para los primeros VLSI... Los ingenieros de Intel sabían poco acerca de arquitectura de computadoras. Su objetivo inmediato era realizar dispositivos programables que sustituirían la lógica aleatoria.”

Maurice Wilkes, Computing Perspectives

“Los días del reloj digital están numerados.”

Tom Stoppard

TEMAS DEL CAPÍTULO

- 2.1 Latches, flip-flops y registros
- 2.2 Máquinas de estado finito
- 2.3 Diseño de circuitos secuenciales
- 2.4 Partes secuenciales útiles
- 2.5 Partes secuenciales programables
- 2.6 Relojes y temporización de eventos

El comportamiento de un circuito combinacional (sin memoria) sólo depende de sus entradas de corriente, no de la historia pasada. En este sentido, un circuito digital secuencial tiene una cantidad finita de memoria, cuyo contenido, determinado por entradas pasadas, afecta el comportamiento de corriente de entrada/salida (I/O). En este capítulo se ofrece un repaso de los métodos para definir y realizar tales circuitos secuenciales mediante elementos de almacenamiento (latches, flip-flops, registros) y lógica combinacional. También se introducen algunos componentes muy útiles que se encuentran en muchos diagramas de este libro. Los ejemplos incluyen archivos de registro, registros de corrimiento y contadores. Al igual que en el capítulo 1, los lectores que tengan problemas para comprender este material deberán consultar cualquiera de los libros de diseño lógico que se mencionan al final del capítulo.

■ 2.1 Latches, flip-flops y registros

El diseño de circuitos secuenciales que exhiben memoria requieren el uso de elementos de almacenamiento capaces de retener información. El elemento de almacenamiento más simple es capaz de retener un solo bit y se puede *establecer* (*set*) en 1 o *restablecer* (*reset*) a 0 a voluntad. El latch SR, que se muestra en la figura 2.1a, es uno de tales elementos. Cuando las entradas *R* y *S* son 0, el latch se

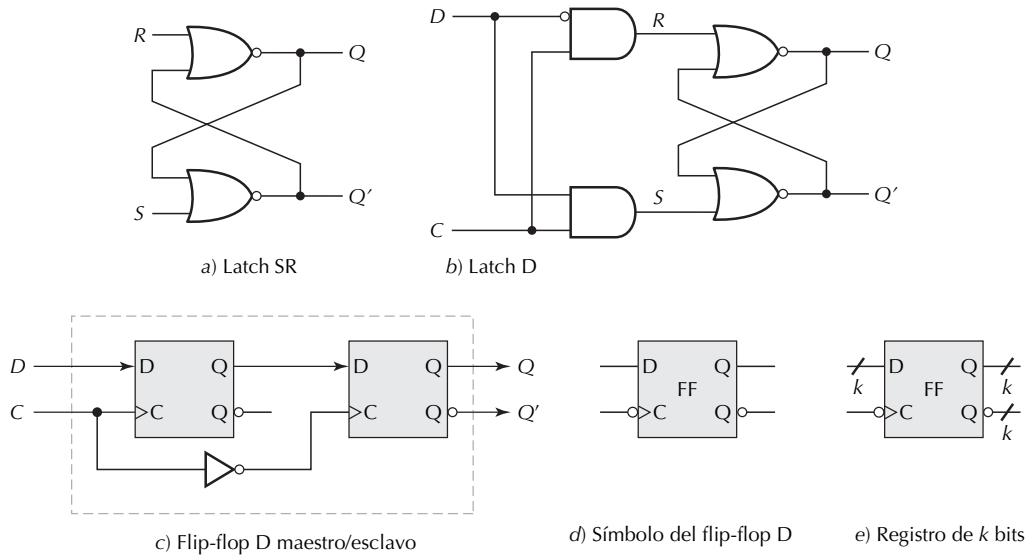


Figura 2.1 Latches, flip-flops y registros.

encuentra en uno de dos estados estables correspondientes a $Q = 0$ ($Q' = 1$) o $Q = 1$ ($Q' = 0$). Postular la entrada R restablece (*reset*) el latch a $Q = 0$, mientras que postular S establece (*set*) el latch a $Q = 1$. En consecuencia, persiste cualquier estado después de que la entrada postulada deja de hacerlo. Agregar dos compuertas AND a un latch SR, como en la figura 2.1b, produce un latch D con entradas de datos (D) y reloj (*clock*, C). Cuando C se postula, la salida Q del latch SR sigue a D (el latch se establece si $D = 1$ y se restablece si $D = 0$). Se dice que el latch está abierto o es *transparente* para $C = 1$. después de que C deja de postularse, el latch SR se cierra y conserva como su salida el último valor de D antes de la pérdida de postulación de C .

Dos latches D se pueden conectar, como en la figura 2.1c, para formar un flip-flop D maestro/esclavo (*master/slave*). Cuando C se postula, el latch maestro se abre y su salida sigue a D , mientras que el latch esclavo se cierra y conserva su estado. La no postulación de C cierra el latch maestro y provoca que su estado se copie en el latch esclavo. La figura 2.1d muestra la notación abreviada de un flip-flop D. La burbuja en la entrada C indica que el nuevo estado del flip-flop tiene efecto en el *extremo negativo* de la entrada clock, o sea cuando el reloj pasa de 1 a 0. Se dice que tal flip-flop es *activado por flanco de bajada* (*negative-edge-triggered*). Es posible construir un flip-flop D *activado por flanco de subida* (*positive-edge-triggered*) al invertir la entrada C de la figura 2.1c. Un arreglo de k flip-flop, todos unidos a la misma entrada de reloj, forma un registro de k bits (figura 2.1e). Puesto que un flip-flop maestro-esclavo conserva su contenido conforme se modifica, es ideal para usar en registros constructivos que con frecuencia se leen y escriben en el mismo ciclo de máquina.

La figura 2.2 muestra cómo los cambios en las entradas D y C afectan la salida Q en un latch D y en un flip-flop D activado por flanco de bajada. En un latch D, la salida Q sigue a D siempre que C se postule y permanezca estable en su último valor cuando C deja de postularse. De este modo, cualquier cambio en Q coincide con un flanco de subida o negativo de C o D ; se involucra un pequeño retardo (*delay*) debido al tiempo de propagación de señal a través de las compuertas. Las flechas en la figura 2.2 indican relaciones causa/efecto. En un flip-flop D, los cambios en Q coinciden con los flancos de bajada de C ; de nuevo se involucra un pequeño retardo. Para evitar metaestabilidad, que causa una operación inadecuada del flip-flop (vea la sección 2.6), el valor de D debe permanecer estable durante

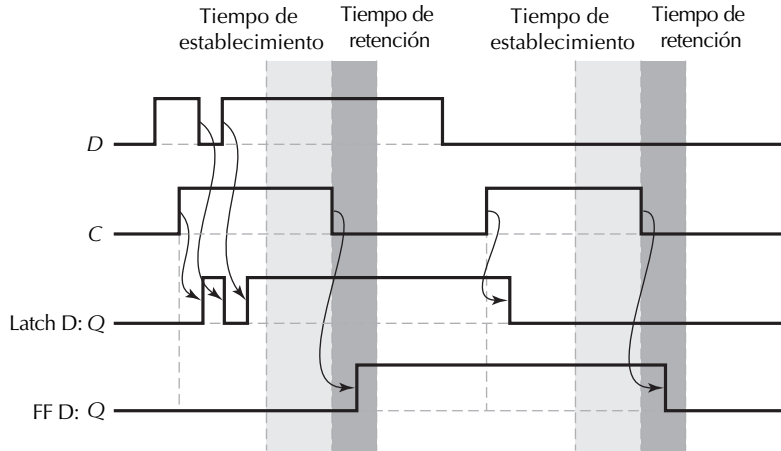


Figura 2.2 Operación de un latch D y de un flip-flop D activado por flanco de bajada.

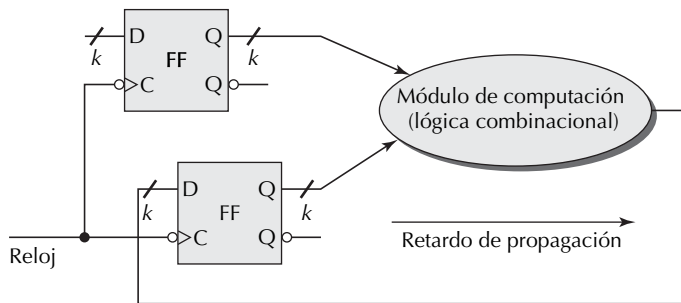


Figura 2.3 Operación registro a registro con flip-flop activados por flanco.

una pequeña ventana antes y después de una caída de flanco de C. El tiempo de estabilidad requerido para D antes de la caída de flanco se conoce como *tiempo de establecimiento* (*setup time*), mientras que después de la caída del flanco se denomina *tiempo de retención* (*hold time*).

La figura 2.3 muestra una interconexión usual de registros y componentes combinacionales en sistemas secuenciales síncronos activados por una señal de reloj. Uno o más registros fuente proporcionan los datos que se usan en un cálculo, el resultado se almacena en un registro de destino. Debido al diseño del registro maestro-esclavo, que conduce a la operación activada por flanco, el registro destino puede ser el mismo que uno de los registros fuente. Conforme las señales que se encuentran operando se propagan a través del módulo de computación y comienzan a afectar el lado de entrada del registro destino, la salida de los registros fuentes permanece estable, ello conduce a la estabilidad necesaria en las entradas del registro destino. En tanto el periodo de reloj sea mayor que la suma del retardo de propagación del latch, el retardo de propagación a través de la lógica combinacional y el tiempo de establecimiento del latch, la operación correcta se garantiza; como regla común, el tiempo de retención se puede ignorar, pues usualmente es más pequeño que el retardo de propagación del latch.

Como consecuencia de que casi todos los circuitos secuenciales que se necesitan en este libro usan flip-flop D, mientras que unos cuantos se basan en flip-flop SR que se derivan fácilmente del latch SR de la figura 2.1a, no se cubrirán otros tipos de flip-flop como el JK y el T. Estos últimos se describen en cualquier libro de diseño lógico, en algunos casos conducen a simplificaciones de diseño; sin em-

bargo, aquí se evitan para mantener el foco en nociones de arquitectura de computadoras, en lugar de en detalles de implementación de circuitos lógicos.

2.2 Máquinas de estado finito

Así como las funciones booleanas y las tablas de verdad son caracterizaciones abstractas de circuitos digitales combinacionales, las *máquinas de estado finito* (o simplemente *máquinas de estado*) y las *tablas de estado* se usan para especificar el comportamiento secuencial de un circuito digital con memoria. Para los circuitos secuenciales simples, en este libro se prefiere la representación gráfica de las tablas de estado, conocidas como *diagramas de estado*. Los circuitos más complejos se describen en forma algorítmica, dado que las representaciones en tabla de estado y en diagrama de estado crecen exponencialmente con el número de variables de estado. Una máquina de estado finito con n bits de almacenamiento puede tener hasta 2^n estados; por tanto, incluso un circuito digital con un solo registro de 32 bits como memoria ya es bastante grande como para describirse en forma de tabla de estado. El proceso de derivar tablas de estado simples se ilustra con un ejemplo.

Ejemplo 2.1: Máquina de estado receptora de monedas Una pequeña máquina expendedora puede dispensar artículos que cuestan cada uno 35 centavos. Los usuarios sólo pueden usar monedas de 25 y diez centavos. La máquina no está programada para regresar cambio, pero está configurada para dispensar el artículo deseado cuando se han depositado 35 o más centavos. Derive una tabla de estado para la unidad receptora de monedas de esta máquina expendedora, si supone que las monedas se depositan, y, por ende, se detectan, una a la vez.

Solución: La figura 2.4 muestra una posible tabla de estado y el diagrama de estado asociado. La etapa inicial es S_{00} . Conforme las monedas se depositan, la unidad se mueve de uno a otro estado para “recordar” la cantidad de monedas que han sido depositadas hasta el momento. Por ejemplo, el depósito de la primera moneda de diez centavos lleva la unidad de S_{00} a S_{10} , donde el nombre de estado S_{10} se elige para representar la cantidad depositada. Al estado S_{35} corresponde a la cantidad depositada que es

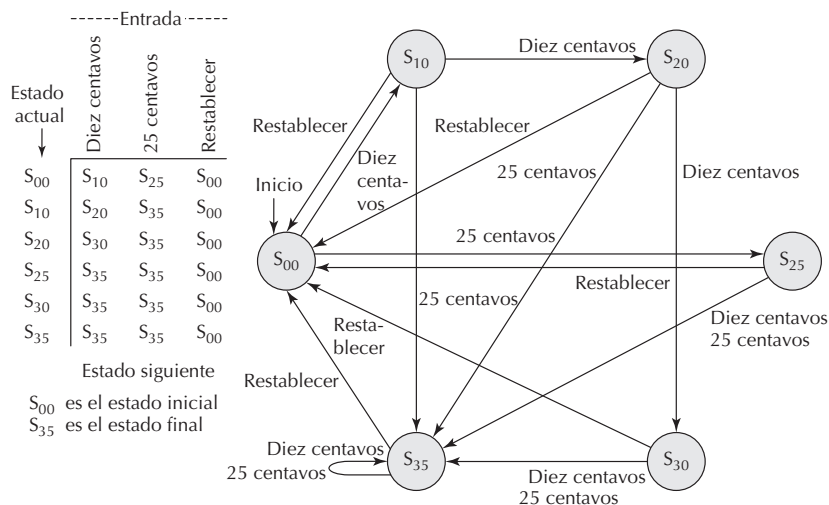


Figura 2.4 Tabla de estado y diagrama de estado para la unidad receptora de monedas de una máquina expendedora.

suficiente para dispensar un artículo. Cuando se llega al estado S_{35} , “venta habilitada”, la máquina de estado finito no experimenta cambio de estado debido a depósitos adicionales de moneda. La máquina tiene tres entradas, que corresponden a tres eventos: inserción de una moneda de diez centavos, inserción de moneda de 25 centavos y restablecer (debido a condiciones de venta no satisfechas, lo que provoca el regreso de monedas, o a venta completa). Como consecuencia de que los estados S_{25} y S_{30} son completamente equivalentes desde el punto de vista del proceso de recepción de monedas (ambas necesitan una moneda adicional de diez o de 25 centavos para permitir la transacción hacia S_{35}), se pueden fusionar para obtener una máquina más simple de cinco estados.

La máquina de estado finito que se representa en la figura 2.4 se denomina *máquina de Moore*, pues su salida se asocia con sus estados. En el estado S_{35} se habilita la entrega del artículo seleccionado, mientras que en todos los otros estados está deshabilitada. En este contexto, en una *máquina de Mealy*, las salidas se asocian con las transiciones entre estados; por tanto, la salida depende tanto del estado presente como de la entrada actual recibida. La figura 2.5 muestra cómo se pueden realizar en hardware las máquinas de Moore y de Mealy. El estado de la máquina se conserva en un registro de l bits que permite hasta 2^l estados. El circuito lógico de estado siguiente produce las señales de excitación requeridas para efectuar un cambio de estado con base en n entradas y l variables de estado, mientras que el circuito lógico de salida produce las m señales de salida de la máquina. Cuando el registro de estado se compone de flip-flops D, el número de señales de excitación de estado siguiente es el mismo que el número de variables de estado, puesto que cada flip-flop D necesita una entrada de datos. Las salidas se derivan sólo con base en variables de estado (máquina de Moore) o de variables de estado y entradas (máquina de Mealy).

2.3 Diseño de circuitos secuenciales

La realización en hardware de circuitos secuenciales, de acuerdo con la estructura general de la figura 2.5, comienza con la selección de los elementos de memoria que se usarán. En este texto se tratará exclusivamente con flip-flops D, de modo que este paso del proceso de diseño está predeterminado. A continuación, los estados se deben codificar con el uso de l variables de estado para un valor adecuado de l . La elección de l afecta el costo de los elementos de memoria. Sin embargo, esto último no significa necesariamente que la meta siempre es minimizar l ; con frecuencia una codificación escasa de estados, con el uso de más variables de estado, conduce a circuitos combinacionales más simples para generar las señales de excitación y las salidas. Una revisión detallada de este proceso de *asignación de estado* y las negociaciones asociadas está más allá del ámbito de la discusión. Observe sólo que el

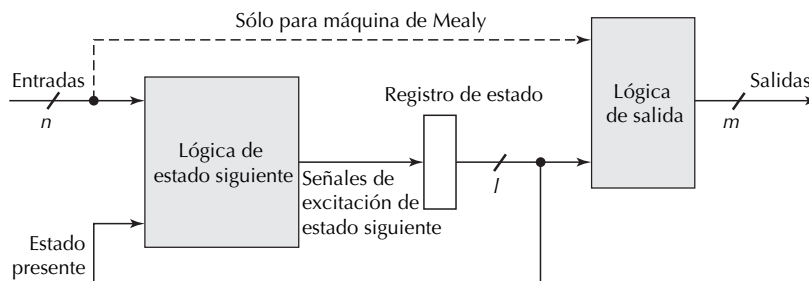


Figura 2.5 Realización en hardware de máquinas secuenciales de Moore y de Mealy.

rango de elecciones, desde la codificación más densa posible, donde 2^l es estrictamente menor que el doble del número de estados, hasta la más escasa, donde l es igual al número de estados y cada código de estado es una cadena de l bits que contiene un solo 1 (a esta situación se le denomina *codificación “1 entre n”, one-hot encoding*). Después de la asignación de estado, son fáciles de derivar las tablas de verdad para los dos bloques combinacionales de la figura 2.5 y, por ende, sus realizaciones en circuito. El proceso completo de diseño se ilustra mediante dos ejemplos.

Ejemplo 2.2: Construcción de un flip-flop JK Diseñe un flip-flop JK a partir de un flip-flop D sencillo y elementos de lógica combinacional. Un flip-flop JK, un elemento de memoria con dos entradas (J y K) y dos salidas (Q y Q'), conserva su estado cuando $J = K = 0$, se restablece a 0 cuando $J = 0$ y $K = 1$, se establece en 1 cuando $J = 1$ y $K = 0$ e invierte su estado (cambia de 0 a 1 o de 1 a 0) cuando $J = K = 1$.

Solución: El enunciado del problema define la tabla de estado para el flip-flop JK (tabla 2.1) que, en esencia, representa la tabla de verdad para una función D de tres variables (J, K, Q). La siguiente entrada de excitación para el flip-flop D se deriva fácilmente de la tabla 2.1: $D = JQ' \vee K'Q$. El circuito resultante se muestra en la figura 2.6. En virtud de que el flip-flop D usado en el diseño es activado por flanco de bajada, igualmente lo es el flip-flop JK resultante.

■ **TABLA 2.1** Tabla de estado para el flip-flop JK que se define en el ejemplo 2.2.

Estado siguiente para Estado actual	$J = 0$ $K = 0$	$J = 0$ $K = 1$	$J = 1$ $K = 0$	$J = 1$ $K = 1$
0	0	0	1	1
1	1	0	1	0

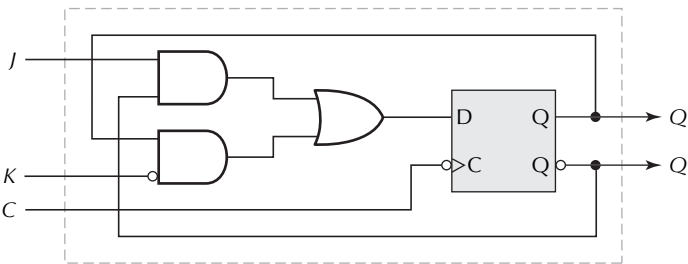


Figura 2.6 Realización en hardware de un flip-flop JK (ejemplo 2.2).

Ejemplo 2.3: Circuito secuencial para una unidad receptora de monedas Considere la tabla de estado y el diagrama de estado para la unidad receptora de monedas de la máquina expendedora derivada en el ejemplo 2.1. Suponga que los estados S_{25} y S_{30} se fusionan en el estado $S_{25/30}$, como se sugirió al final de la solución para el ejemplo 2.1. Diseñe un circuito secuencial para implementar la máquina de cinco estados resultante con el uso de flip-flops D con entradas tipo restablecer asíncronas separadas.

Solución: Existen cinco estados, sólo uno permite completar una venta. Se necesitan al menos tres bits para codificar los cinco estados. Puesto que sólo el estado S_{35} permite que la venta proceda, parece natural distinguir dicho estado al codificarlo como $Q_2Q_1Q_0 = 1xx$, donde Q_2 es la salida “habilitar venta”. Para los estados restantes, se pueden usar las asignaciones siguientes: 000 para S_{00} (pues es el estado restablecer), 001 para S_{10} , 010 para S_{20} y 011 para $S_{25/30}$. Sean q y d las entradas que, cuando se postulan para un ciclo de reloj, indican la detección de una moneda de 25 centavos y otra de diez centavos, respectivamente. Como consecuencia de que las monedas se insertan y detectan una a la vez, $q = d = 1$ representa una condición del tipo “no importa”. Estas elecciones conducen a una tabla de estado codificada (tabla 2.2), que, en esencia, constituye la tabla de verdad para tres funciones (D_2, D_1, D_0) de cinco variables (q, d, Q_2, Q_1, Q_0). Las siguientes entradas de excitación se derivan fácilmente para los flip-flops D: $D_2 = Q_2 \vee qQ_0 \vee qQ_1 \vee dQ_1Q_0$, $D_1 = q \vee Q_1 \vee dQ_0$, $D_0 = q \vee d'Q_0 \vee dQ_0'$. El circuito resultante se muestra en la figura 2.7.

■ **TABLA 2.2** Tabla de estado para una unidad receptora de monedas después de elegir la asignación de estado del ejemplo 2.3.

Estado siguiente para Estado actual	$q = 0$ $d = 0$	$q = 0$ $d = 1$	$q = 1$ $d = 0$	$q = 1$ $d = 1$
$S_{00} = 000$	000	001	011	xxx
$S_{10} = 001$	001	010	1xx	xxx
$S_{20} = 010$	010	011	1xx	xxx
$S_{25/30} = 011$	011	1xx	1xx	xxx
$S_{35} = 1xx$	1xx	1xx	1xx	xxx

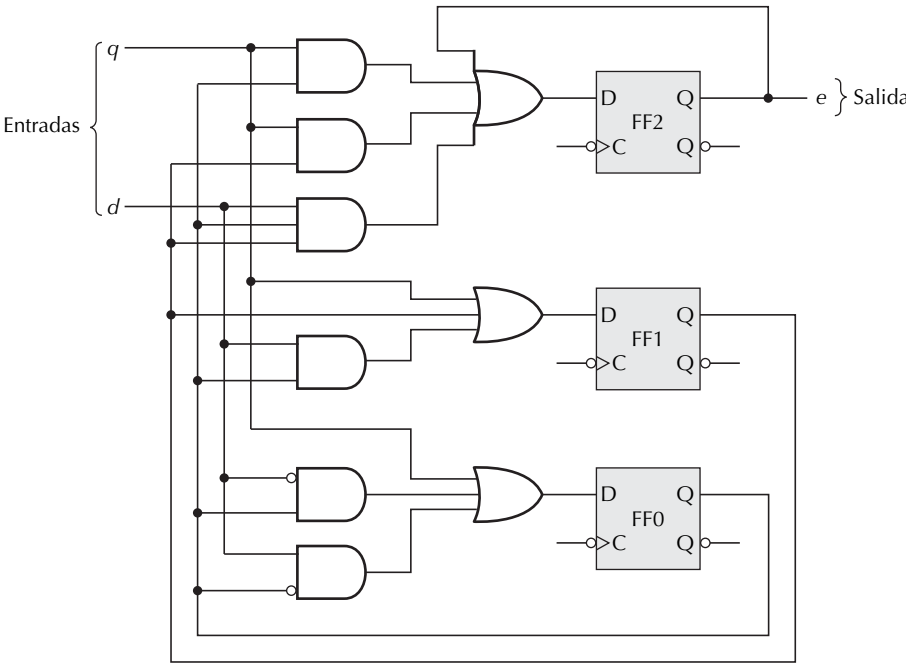


Figura 2.7 Realización de hardware de una unidad receptora de monedas (ejemplo 2.3).

■ 2.4 Partes secuenciales útiles

Un registro representa un arreglo de flip-flops con entradas de datos individuales y señales de control comunes. En la construcción de sistemas digitales se usan comúnmente ciertos tipos especiales de registros. Por ejemplo, un registro de corrimiento se puede cargar con nuevo o su antiguo contenido corrido hacia la derecha o a la izquierda. Desde el aspecto conceptual, un registro de corrimiento se puede construir de un registro ordinario conectado a un multiplexor. En la figura 2.8 se muestra un registro capaz de correr a la izquierda un solo bit. Cuando el registro es temporizado, una nueva palabra de datos o la versión corrida a la izquierda de la palabra almacenada en registro sustituye su antiguo contenido. Al usar un multiplexor más grande y señales de control adecuadas, se pueden acomodar otros tipos de corrimiento. Los corrimientos multibit se acomodan o al realizar varios corrimientos de bit sencillo durante ciclos de reloj sucesivos (lo que es más bien lento) o al usar un circuito combinatorial especial que pueda correr varias cantidades. El diseño de tal *registro de corrimiento en cilindro* (*barrel shifter*), que se usa principalmente en la alineación y normalización de operandos y resultados de punto flotante, se discutirá en el capítulo 12.

Tal como un registro representa un arreglo de flip-flops (figura 2.1e), un *archivo de registro* constituye un arreglo de registros (figura 2.9). El índice o dirección de un registro se usa para leer de él o escribir en él. Para un archivo de registro con 2^k registros, la dirección de registro es un número binario de h bits. En virtud de que muchas operaciones requieren más de un operando, los archivos de registro casi siempre son *multiportados* (*multiported*), ello significa que son capaces de proporcionar las palabras de datos almacenadas en registros múltiples a la vez, mientras que, al mismo tiempo, escribe en uno o más registros. Los registros contruidos a partir de flip-flops maestro-esclavo activados por flanco de bajada se pueden leer y modificar en el mismo ciclo de reloj, y los cambios en el contenido suceden en el siguiente ciclo de reloj. Por tanto, una operación como $B \leftarrow A + B$, donde A y B se almacenan en registros, se puede ejecutar en un solo ciclo de reloj, pues al escribir el nuevo valor de B no se interfiere con la lectura de su valor presente (vea la figura 2.3).

Un *FIFO* es un archivo de registro especial *first-in, first-out* (primero en entrar, primero en salir) a cuyos elementos se ingresa en el mismo orden en que se colocaron. En consecuencia, un FIFO no necesita una entrada de dirección; más bien, está equipado con un puerto de lectura, un puerto de escritura y señales indicadoras especiales designadas “FIFO empty” (FIFO vacío) y “FIFO full” (FIFO lleno). Como se muestra en la figura 2.9c, las señales de habilitación de escritura y lectura para un FIFO se llaman “push” (meter) y “pop” (sacar), respectivamente, lo que indica si los datos se meten al FIFO o se sacan de él. Un FIFO representa una *cola* (*queue*) implementada con hardware con un tamaño máximo fijo.

Un dispositivo SRAM (*static random-access memory*, memoria de acceso aleatorio estática) es muy parecido a un archivo de registro, excepto que usualmente es de un solo puerto y mucho más grande en capacidad. Un chip SRAM de $2^h \times g$ (figura 2.10) recibe un sumador de h bits como entrada y propor-

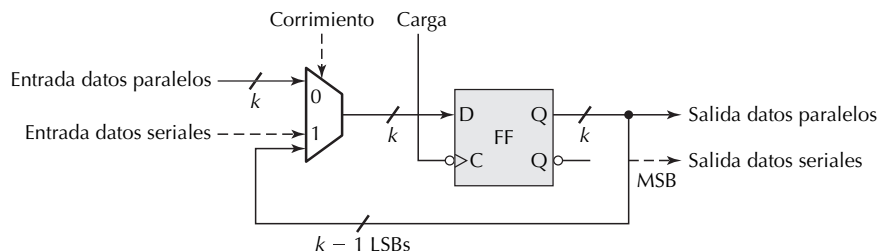


Figura 2.8 Registro con corrimiento izquierdo de un solo bit y capacidades de carga en paralelo. Para corrimiento izquierdo lógico, los datos seriales en línea se conectan a 0.

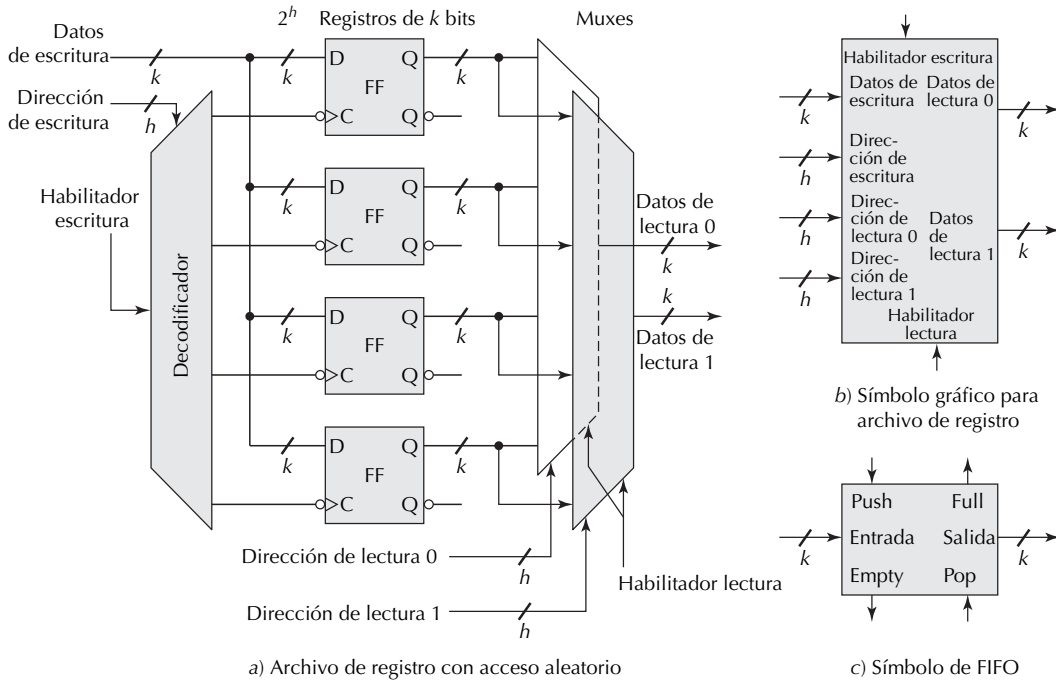


Figura 2.9 Archivo de registro con acceso aleatorio y FIFO.

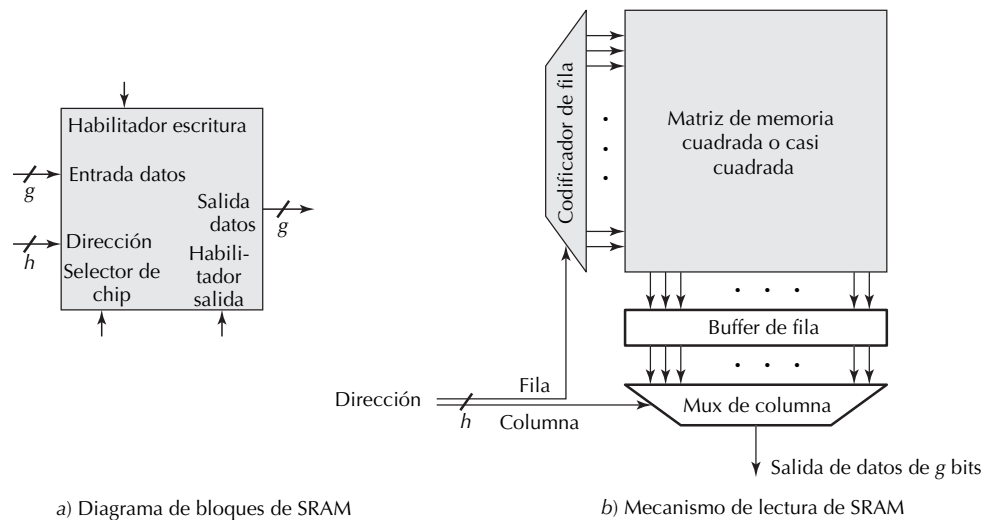


Figura 2.10 La memoria SRAM constituye un gran archivo de registro de puerto sencillo.

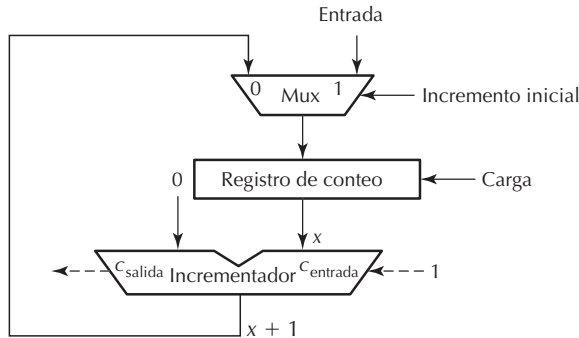


Figura 2.11 Contador binario síncrono con capacidad de inicialización.

ciona g bits de datos como salida durante una operación de lectura. Para una operación de escritura, en la ubicación seleccionada se escriben g bits de datos de entrada. Los datos de entrada y salida usualmente comparten los mismos pines, en virtud de que nunca se usan al mismo tiempo. La señal de habilitación de escritura tiene la misma participación que en los archivos de registro. La señal de habilitación de salida controla un conjunto de buffers tres estados que proporcionan los datos de salida; por ende, permiten conectar algunos chips a un solo conjunto de líneas de salida o bus. Para formar una memoria con palabras de k bits, donde $k > g$, se conectan k/g de tales chips en paralelo, todos ellos reciben las mismas señales de dirección y control y cada uno proporciona o recibe g bits de la palabra de datos de k bits (en el raro evento de que g no divida k , se usan $\lceil k/g \rceil$ chips). De igual modo, para formar una memoria con 2^m ubicaciones, donde $m > h$, se usan 2^{m-h} filas de chips, cada una con k/g chips, donde se usa un decodificador externo de $(m-h)$ a 2^{m-h} para seleccionar la fila de chips que tendría postulada su señal de selección de chip.

La tecnología SRAM se usa para implementar pequeñas memorias rápidas ubicadas cerca del procesador (con frecuencia en el mismo chip). La memoria principal más grande de una computadora se implementa en tecnología DRAM (*dynamic random-access memory*, memoria de acceso aleatorio dinámica), que se discutirá en el capítulo 17. En este contexto, DRAM requiere el uso de un transistor para almacenar un bit de datos, mientras que SRAM necesita muchos transistores por cada bit. Esta diferencia hace DRAM más densa y barata, pero también más lenta, que SRAM. Las *memorias de sólo lectura* (ROM: *read-only memory*) también se usan comúnmente en el diseño de sistemas digitales. Se puede usar un ROM de $2^h \times g$ para realizar g funciones booleanas de h variables y, por ende, se observa como una forma de lógica programable. El contenido de ROM se puede cargar permanentemente en el momento de fabricación o “programarse” en la memoria mediante dispositivos especiales de programación de ROM. En este caso, se tiene un dispositivo *ROM programable* (PROM). Una *PROM borrable* (EPROM) se puede programar más de una vez.

Un *contador hacia arriba* se construye de un registro y un incrementador, como se muestra en la figura 2.11. De igual modo, un *contador hacia abajo* se compone de un registro y un decrementador. En ausencia de calificación explícita, el término “contador” se usa para un contador arriba. Un *contador arriba/abajo* puede contar o arriba o abajo mediante el control de una señal de dirección. El diseño de contador que se muestra en la figura 2.11 es adecuado para la mayoría de las aplicaciones. Se puede hacer más rápido si se usa un incrementador rápido con característica de anticipación de acarreo (*carry-lookahead*) similar al que se usa en los sumadores rápidos, que se discutirán en el capítulo 10. Si todavía se requiere más rapidez, el contador se puede dividir en bloques. Un corto bloque inicial (por ejemplo, tres bits de ancho) puede trabajar fácilmente con las rápidas señales entrantes. Los bloques cada vez más anchos a la izquierda del bloque inicial no necesitan ser tan rápidos porque ellos se ajustan cada vez con menos frecuencia.

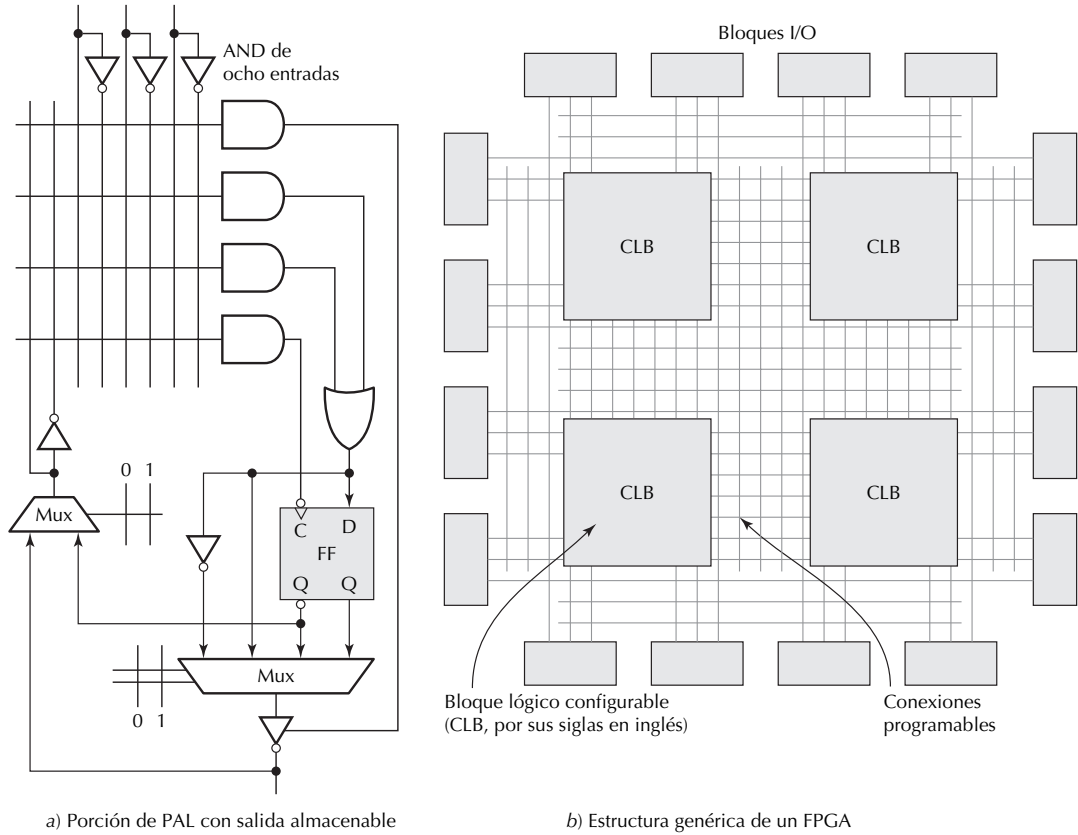


Figura 2.12 Ejemplos de lógica secuencial programable.

2.5 Partes secuenciales programables

Las partes secuenciales programables consisten de arreglos programables de compuertas con elementos de memoria estratégicamente colocados para retener datos de un ciclo de reloj al siguiente. Por ejemplo, una forma de PAL con elementos de memoria usada comúnmente tiene una estructura similar a la de la figura 1.13b, pero cada salida de compuerta OR se puede almacenar en un flip-flop y la salida del dispositivo es elegible de entre la salida de la compuerta OR, su complemento y las salidas de flip-flop (figura 2.12a). En este sentido, la salida de la compuerta OR o la salida del flip-flop se pueden alimentar de vuelta en el arreglo AND a través de un multiplexor 2 a 1. Las tres señales que controlan los multiplexores de la figura 2.12a se pueden vincular al 0 o 1 lógicos mediante conexiones programables.

Los circuitos programables similares al que se muestra en la figura 2.12a ofrecen todos los elementos necesarios para implementar una máquina secuencial (figura 2.5) y también se pueden usar como partes combinacionales si se desea. Tales dispositivos tienen dos partes diferentes. El *arreglo de compuertas*, que, en esencia, representa una PAL o un PLA como los que se muestran en la figura 1.13, se proporciona por su capacidad de realizar funciones lógicas deseadas. La *macrocelda de salida*, que contiene uno o más flip-flop, multiplexores, buffers tres estados o inversores, forma las salidas requeridas con base en valores derivados en el ciclo actual y aquellos derivados en ciclos oportunos y almacenados en los diversos elementos de memoria.

Lo último en flexibilidad lo ofrecen los *arreglos de compuertas programables por campo* (FPGA, *field-programmable gate array*), que se muestran en forma simplificada en la figura 2.12b, compuestos por gran número de bloques lógicos configurables (CLB) en el centro, rodeados por bloques I/O en los extremos. Conexiones programables llenan los espacios entre los bloques. Cada CLB es capaz de realizar una o dos funciones lógicas arbitrarias de un pequeño número de variables y también tiene uno o más elementos de memoria. Cada bloque I/O es similar a la macrocelda de salida en la mitad inferior de la figura 2.12a. Grupos de CLB y de bloques I/O se pueden ligar mediante interconexiones programables para formar sistemas digitales complejos. Los elementos de memoria (usualmente SRAM) que conservan el patrón de conectividad entre celdas se puede inicializar a partir de algunos ROM antes de comenzar a definir la funcionalidad del sistema. También se pueden cargar con nuevos valores en cualquier momento para efectuar *reconfiguración de tiempo de ejecución*. Paquetes de software especiales, proporcionados por los fabricantes de FPGA y proveedores independientes, permiten el mapeo automático en FPGA de funcionalidad de hardware especificada algorítmicamente. En las referencias al final del capítulo observa una discusión más detallada acerca de los FPGA y de otras partes programables secuenciales.

■ 2.6 Relojes y temporización de eventos

Un reloj constituye un circuito que produce una señal periódica, usualmente a una frecuencia o tasa constante. El inverso de la tasa de reloj es el periodo del reloj. Por ejemplo, un reloj de un gigahertz (GHz) tiene una frecuencia de 10^9 y un periodo de $1/10^9 \text{ s} = 1 \text{ ns}$. Usualmente, la señal de reloj está en 0 o 1 durante casi la mitad del periodo de reloj (figura 2.13). La operación de un circuito secuencial síncrono es gobernada por un reloj. Con los flip-flop (FF) activados por flanco, la operación correcta del circuito que se muestra en la figura 2.13 se puede asegurar al propiciar que el periodo del reloj sea lo suficientemente largo para acomodar el peor retardo posible a través de lógica combinacional, t_{comb} , mientras que todavía queda suficiente tiempo para el tiempo de establecimiento (*setup*) de FF2. Como consecuencia de que cualquier cambio en FF1 no es instantáneo, sino que requiere una cantidad de tiempo conocida como *tiempo de propagación*, t_{prop} , se llega al siguiente requisito:

$$\text{Periodo de reloj} \geq t_{\text{prop}} + t_{\text{comb}} + t_{\text{setup}} + t_{\text{skew}}$$

El término adicional, t_{skew} , para *desfase de reloj*, muestra la posibilidad de que, debido a retardos de propagación de señal y ciertas anomalías, la señal de reloj que controla FF2 pueda llegar ligeramente adelantada de la de FF1, y, por ende, acorta efectivamente el periodo de reloj. Observe que el tiempo de retención del flip-flop está ausente de la desigualdad anterior porque casi siempre es menor que t_{prop} (es

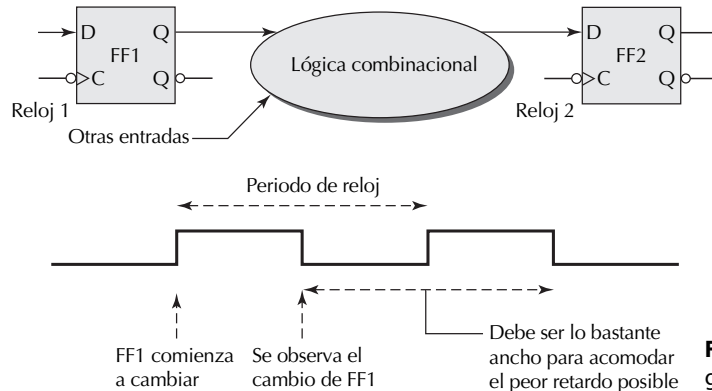


Figura 2.13 Determinación de la longitud requerida del periodo de reloj.

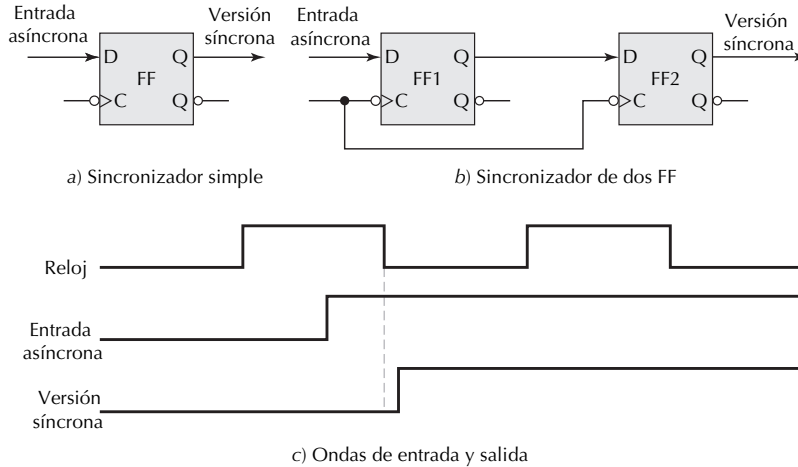


Figura 2.14 Los sincronizadores se usan para evitar problemas de temporización que podrían surgir de cambios no temporizados en señales asíncronas.

decir, durante t_{prop} , la salida de un flip-flop permanece en su valor previo; en consecuencia, satisface el requisito de tiempo de retención).

Una forma de permitir mayor tasa de reloj y mayor rendimiento de trabajo, es usar *segmentación (encauzamiento)*. En modo encauzado, la lógica combinacional de la figura 2.13 se divide en etapas y elementos de almacenamiento insertados entre etapas para retener resultados de cómputo parciales. De esta forma, un nuevo cómputo puede comenzar tan pronto haya la seguridad de almacenar los resultados de la primera etapa. Sin embargo, esta técnica afecta sólo al término t_{comb} en la desigualdad precedente. El periodo de reloj todavía debe acomodar los otros tres términos. Por esta razón, existe un límite más allá del cual sumar más etapas del encauzamiento no será satisfactorio.

Implícita en la desigualdad anterior está la suposición de que las señales no cambian dentro del periodo de reloj así determinado. En otras palabras, si la restricción apenas se satisface, como con frecuencia es el caso cuando se intenta usar la tasa de reloj más rápida posible para maximizar el rendimiento, entonces cualquier cambio en los valores de señal dentro del periodo de reloj conducirá a una violación de temporización. Puesto que muchas señales de interés pueden provenir de unidades no gobernadas por el mismo reloj (éstas se conocen como *entradas asíncronas*), sus variaciones están más allá del control. Por esta razón, tales señales pasan a través de circuitos especiales conocidos como *sincronizadores*, cuya función es garantizar la estabilidad de la señal por la duración de tiempo requerida. La figura 2.14a muestra cómo se puede usar un solo flip-flop para sincronizar una entrada asíncrona. Cualquier cambio en la entrada asíncrona se observa sólo inmediatamente después del siguiente borde de reloj negativo. Sin embargo, existe una posibilidad de que el cambio ocurra muy cerca del siguiente borde de reloj negativo (figura 2.14c), que conduce a una condición de metaestabilidad, donde la señal observada no es ni 0 ni 1; peor todavía, la señal puede aparecer como 0 durante algunas unidades y como 1 durante otras. Por esta razón se prefieren los sincronizadores de 2 FF (figura 2.14b). Aunque éstos no eliminan la metaestabilidad, hacen muy improbable que los problemas prácticos sean evitados.

Una operación poco más rápida resulta si se usan latches y temporización sensible al nivel en lugar de flip-flops activados por flanco (con el propósito de observar esto último, compare los circuitos latch y flip-flop de la figura 2.1). Con la operación sensible a nivel, en lugar de cambios abruptos que coincidan con flancos de reloj, un latch permanece abierto durante todo el tiempo que la señal del reloj es alta. Ello puede conducir a problemas si dos latches sucesivos siempre están abiertos al mismo tiempo.

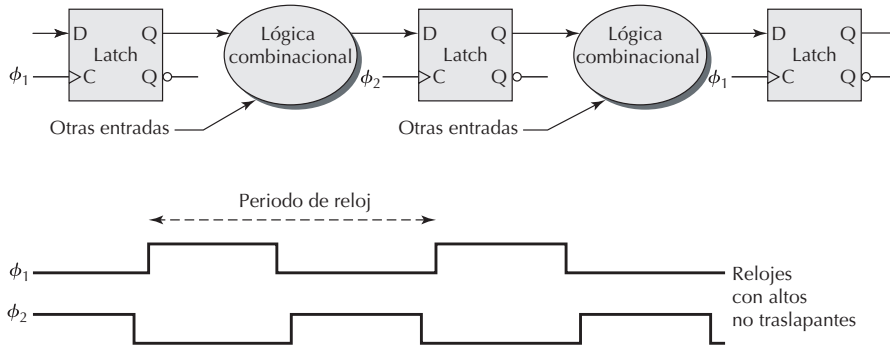


Figura 2.15 Cronometrado en dos fases con señales de reloj no traslapantes.

Por esta razón, la temporización sensible al nivel con frecuencia se usa en conjunción con *cronometrado en dos fases*. En este esquema, que se muestra en la figura 2.15, se usan dos señales de reloj no traslapantes, ϕ_1 y ϕ_2 (para fase 1 y fase 2) se usan para controlar latches sucesivos en la trayectoria de computación. Lo anterior ayuda a garantizar que, cuando un latch está abierto, el siguiente latch en el camino siempre está cerrado y viceversa. Como resultado, las señales se propagan sólo de un latch al siguiente, como en el método activado por flanco. Por simplicidad, en este texto siempre se considerará que los circuitos tienen temporización activada por flanco, a sabiendas de que cualquiera de tales diseños se puede convertir a operación sensible a nivel mediante cronometrado en dos fases.

PROBLEMAS

2.1 Formas alternativas del latch D

- Con el hecho de que una compuerta OR es reemplazable por una compuerta NAND con entradas invertidas, demuestre que el latch D se puede construir con cuatro compuertas NAND y un inversor.
- Demuestre que el inversor en el diseño de la parte a) se puede remover, dejando sólo cuatro compuertas NAND, al alimentar la salida de una de las otras compuertas a la compuerta que tiene el inversor en una entrada.

2.2 Latch D frente a flip-flop D

Caracterice, en términos de valores y temporización, todos los pares de señal de entrada, D y C , que, cuando se aplican a un latch D y un flip-flop D, conducen exactamente a la misma salida observada Q .

2.3 Otros tipos de flip-flop

El flip-flop D maestro-esclavo activado por flanco de bajada no es sino uno de los muchos que se pueden

construir. Cada uno de ellos se puede describir mediante una tabla característica que relaciona su cambio de estado con cambios en su(s) línea(s) de entrada. El caso del flip-flop JK se analizó en el ejemplo 2.2. Demuestre cómo cada uno de los siguientes tipos de flip-flop se puede construir a partir de un flip-flop D y una cantidad mínima de lógica combinacional.

- Un flip-flop SR (*set/reset*) que conserva su estado para $S = R = 0$, se establece en 1 para $S = 1$, se restablece a 0 para $R = 1$ y se comporte de manera impredecible para $S = R = 1$. Puede usar la última combinación de valores de entrada como del tipo “no importa”, si supone que nunca ocurrirán en la operación.
- Un flip-flop T (*toggle*), con una entrada T , que conserva su estado para $T = 0$ e invierte su estado para $T = 1$.
- Un flip-flop D con entradas *preset* y *clear* separadas que, en esencia, tiene los mismos efectos que las entradas S y R del flip-flop SR definido en la parte a). Cuando se postulan o S o R , el flip-flop se comporta como un flip-flop SR; de otro modo, se comporta

como un flip-flop D. El caso $S = R = 1$ se considera una condición del tipo “no importa”.

2.4 Otros tipos de flip-flop

Suponga la disponibilidad de los flip-flops T definidos en la parte b) del problema 2.3. Demuestre cómo cada uno de los siguientes tipos de flip-flop se pueden construir a partir de un flip-flop T y una cantidad mínima de lógica combinacional.

- Un flip-flop SR (*set/reset*) que conserva su estado para $S = R = 0$, se establece en 1 para $S = 1$, se restablece a 0 para $R = 1$ y se comporta de manera impredecible para $S = R = 1$. Puede usar la última combinación de valores de entrada como un tipo “no importa”, si supone que nunca ocurrirá en la operación.
- Un flip-flop JK, como se define en el ejemplo 2.2.
- Un flip-flop D con entradas *preset* y *clear* separada que, en esencia, tiene los mismos efectos que las entradas S y R del flip-flop SR definido en la parte a). Cuando se postulan o S o R , el flip-flop se comporta como un flip-flop SR; de otro modo, se comporta como un flip-flop D. El caso $S = R = 1$ se considera una condición del tipo “no importa”.

2.5 Circuitos secuenciales

Identifique el estado de un circuito secuencial construido con h flip-flops D, con salidas designadas Q_{h-1}, \dots, Q_1, Q_0 y entradas etiquetadas D_{h-1}, \dots, D_1, D_0 con el número binario $(Q_{h-1} \dots Q_1 Q_0)_{\text{dos}}$. Para cada uno de los siguientes circuitos secuenciales, definidos por las ecuaciones lógicas para las entradas del flip-flop D, dibuje un diagrama lógico, construya una tabla de estado y derive un diagrama de estado. En estas ecuaciones, a y b son entradas y z es la salida. En cada caso, especifique si el circuito representa una máquina de Mealy o de Moore.

- $D_1 = a(Q_1 \vee Q_0)$, $D_0 = aQ'_1$, $z = a'(Q_1 \vee Q_0)$
- $D_1 = a'b \vee aQ_1$, $D_0 = a'Q_0 \vee aQ_1$, $z = Q_0$
- $D_2 = a' \oplus Q_0 \oplus Q_1$, $D_1 = Q_2$, $D_0 = Q_1$

2.6 Diseño de un conmutador 2×2 programable

Diseñe un conmutador 2×2 programable. El conmutador tiene entradas de datos a y b , una señal de control “cruzada” almacenada c y salidas de datos x y y . Tiene dos modos, programación y operación normal, que se distinguen por la señal externa de control *Prog*. En el

modo programación, la entrada de datos a se almacena en el flip-flop control, donde b actúa como una señal de habilitación de carga. La operación normal del switch depende del valor de c . Cuando $c = 0$, a se conecta a x y b a y . Cuando $c = 1$, a se conecta a y y b a x (es decir, las entradas se cruzan).

2.7 Diseño de un conmutador de clasificación 2×2

Un switch de clasificación 2×2 tiene entradas numéricas binarias sin signo a y b que ingresan bits en forma serial, comenzando con el bit más significativo (MSB, por sus siglas en inglés) y salidas de datos x y y producidas en la misma forma un ciclo de reloj después de la llegada de las entradas, ello se indica mediante la señal de control a_{MSB} . La salida x es la más pequeña de las dos entradas, mientras que y es la más grande de las dos entradas. Por ende, el switch clasifica sus entradas en orden no descendente.

- Presente el diseño completo de un switch de clasificación 2×2 . *Sugerencia:* Use dos muxes y dos FF que inicialmente están *reset*; un FF está en *set* cuando se establece que $a < b$ y el otro está *set* para $b < a$.
- Demuestre cómo se pueden interconectar tres de tales switches para formar un switch de clasificación 3×3 . ¿Cuál es la latencia entrada a salida de este switch?
- Subraye los cambios que se necesitarían en el conmutador 2×2 si cada entrada numérica estuviese precedida por su bit de signo (1 para negativo, 0 para positivo). No es necesario presentar un diseño completo.

2.8 Diseño de contadores

Presente el diseño completo de un contador de cuatro bits que se puede inicializar a su primer estado y cuente cíclicamente:

- De acuerdo con la secuencia 1000, 0100, 0010, 0001 (contador de anillo).
- De acuerdo con la secuencia 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001 (contador Johnson)
- De 0 a 9 (contador de décadas)
- Por tres, módulo 16; por tanto, el contador procede de acuerdo con la secuencia 0, 3, 6, 9, \dots , regresa a 2 después de llegar al 15, y procede a 1 después del 14.

2.9 Registro de corrimiento con carga paralela y *reset*

Con multiplexores 4 a 1 y latches D, diseñe un registro de corrimiento con una señal de control de dos bits interpretada como sigue: 00, retiene datos; 01, carga datos de entrada en paralelo; 10, corre a la izquierda el contenido, desecha el bit de la extrema izquierda y establece el bit de la extrema derecha al valor suministrado en una entrada serial; 11, restablece todos los bits a 0.

2.10 Extensión de la unidad receptora de monedas

Extienda la unidad receptora de monedas del ejemplo 2.3, de modo que también se acepten monedas de cinco centavos. Todas las demás especificaciones de la máquina expendedora y su recepción de monedas permanecen iguales.

2.11 Otra máquina expendedora

Una máquina expendedora vende tres artículos que cuestan 50 centavos, 75 centavos y un dólar. Sólo acepta monedas de 25 centavos o billetes de dólar y regresa hasta dos monedas de 25 centavos en cambio cuando la cantidad insertada supera el costo del artículo seleccionado. Por simplicidad, suponga que, siempre que se regresa cambio, está disponible el número requerido de monedas de 25 centavos. Establezca todas sus suposiciones.

2.12 Cerradura con combinación digital

Una cerradura segura abre si los dos botones A y B se oprimen en la secuencia AABABA (una verdaderamente segura tendría más botones o un código más largo). Oprimir tres veces seguidas B restablece la cerradura a su estado inicial. Oprimir A cuando se debió oprimir B hace sonar una alarma. Oprimir cualquier botón después de abierta la cerradura restablecerá la cerradura.

- Construya el diagrama de estado de la cerradura con combinación digital como una máquina de Mealy.
- Implemente la máquina secuencial definida en la parte a).
- Construya el diagrama de estado de la cerradura con combinación digital como una máquina de Moore.
- Implemente la máquina secuencial definida en la parte c).

2.13 Implementación de máquina secuencial

Cuando el comportamiento de una máquina secuencial se define en términos de una corta historia de entradas pasadas, es posible diseñar la máquina directamente con un registro de corrimiento que retiene la corta historia (estado). En algunos casos, derivar una tabla de estado o un diagrama de estado puede ser problemático en vista de la gran cantidad de estados.

- Use este método y diseñe una máquina secuencial de Mealy con una entrada binaria y una salida binaria. La salida debe ser 1 siempre que los últimos seis bits de entrada, cuando se interpretan como número binario con su bit menos significativo (LSB, por sus siglas en inglés) llegando primero, representan un número primo.
- Repita la parte a) para una máquina secuencial de Moore.

2.14 Implementación de máquina secuencial

Cuando el comportamiento de una máquina secuencial se define en términos de la multiplicidad de ciertos valores o eventos de entrada, es posible diseñar la máquina directamente con uno o más contadores que retienen las diversas cuentas de interés (estado). En algunos casos, derivar una tabla de estado o diagrama de estado puede ser engorroso en vista de la gran cantidad de estados.

- Use este método y diseñe una máquina secuencial de Mealy con una entrada binaria y una salida binaria. La salida debe ser 1 siempre que el número de 1 recibidos hasta el momento es $8m + 1$ para algún entero m .
- Repita la parte a) para una máquina secuencial de Moore.

2.15 Construcción de registros de corrimiento y contadores más grandes

- Explique cómo construiría un registro de corrimiento de 32 bits, de acuerdo con dos registros de corrimiento de 16 bits.
- Repita la parte a) para un contador de hasta 32 bits, dados dos contadores de hasta 16 bits.
- Repita la parte b) para contadores arriba/abajo.

2.16 Derivación de diagramas de estado

Un sumador binario de bits seriales recibe dos números sin signo comenzando desde sus bits menos significativos y produce la suma en la misma forma.

- a) Visualice este sumador como una máquina de Mealy, construya su diagrama de estado (necesita dos estados).
 - b) Implemente la máquina de Mealy de la parte a) con un flip-flop D.
 - c) Repita la parte a) para una máquina secuencial de Moore (se requieren cuatro estados).
 - d) Implemente la máquina de Moore de la parte c) con dos flip-flop D.
- a) Diseñe el circuito comparador y suponga que las entradas de datos llegan comenzando con sus bits más significativos y que en el ciclo de reloj se postula una señal de *reset* antes de que lleguen los datos.
 - b) Repita la parte a) para las entradas suministradas del extremo menos significativo.
 - c) Discuta brevemente cómo se puede modificar el diseño de las partes a) o b) para operar con entradas en complemento a 2.

2.17 Controlador de semáforo simple

Un semáforo en la intersección de las calles norte-sur (NS) y este-oeste (EO) pasa por el siguiente ciclo de estados: ambos rojos (5 s), NS verde (30 s), NS amarillo (5 s), ambos rojos (5 s), EO verde (30 s), EO amarillo (5 s). Para temporización, está disponible una señal de reloj de 0.2 Hz (un pulso de reloj cada 5 s). Ambas calles están equipadas con sensores que detectan la presencia de un automóvil cerca de la intersección. Siempre que exista un carro cerca de la intersección en la calle que en ese momento tenga la luz verde, el interruptor avanza y la luz verde inmediatamente cambia a amarillo. Diseñe un circuito secuencial para este controlador de semáforo.

2.18 Comparador de números de bit seriales

Un comparador de bits seriales para números binarios no signados recibe un bit de datos de entrada de cada uno de los dos operandos x y y , ello produce las señales de salida G ($x > y$ hasta aquí) y E ($x = y$ hasta aquí).

2.19 Archivo de registro

Se tienen dos archivos de registro y cada uno contiene 32 registros que tienen 32 bits de ancho.

- a) Muestre cómo usar estos dos archivos de registro, y elementos lógicos adicionales según se necesiten, para formar un archivo de registro con 64 registros de 32 bits de ancho.
- b) Repita la parte a) para un archivo de registro con 32 registros de 64 bits de ancho.
- c) Repita la parte a) para un archivo de registro con 128 registros de 16 bits de ancho.

2.20 Registro de corrimiento bidireccional

Aumente el registro de corrimiento de la figura 2.8 con elementos lógicos adicionales según se necesiten, de modo que, dependiendo del estado de una entrada de control, Izquierda/Derecha, se pueden realizar corrimientos de los bits individuales izquierdo y derecho.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Brow00] Brown, S. y Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill, 2000.
- [Erce99] Ercegovac, M. D., T. Lang y J. H. Moreno, *Introduction to Digital Systems*, Wiley, 1999.
- [Haye93] Hayes, J. P., *Introduction to Digital Logic Design*, Addison-Wesley, 1993.
- [Katz94] Katz, R. H., *Contemporary Logic Design*, Benjamin/Cummings, 1994.
- [Tocc01] Tocci, R. J. y N. S. Widmer, *Digital Systems: Principles and Applications*, 8a. ed., Prentice Hall, 2001.
- [Wake01] Wakerly, J. F., *Digital Design: Principles and Practices*, Prentice Hall, 3a. ed. actualizada, 2001.
- [WWW] Nombres Web de algunos fabricantes de dispositivos lógicos programables secuenciales: actel.com, altera.com, atmel.com, cypress.com, latticesemi.com, lucent.com, philips.com, quicklogic.com, vantis.com, xilinx.com.

TECNOLOGÍA DE SISTEMAS DE COMPUTACIÓN

“El de la izquierda será obsoleto dentro de ocho meses, mientras que, por sólo 300 dólares más, usted podrá tener el modelo de la derecha que no será obsoleto durante todo un año.”

Explicación de un vendedor a un comprador aturdido, quien intenta decidir cuál de dos computadoras comprar (leyenda de una caricatura de un artista desconocido)

“He descubierto que muchas personas se dan el crédito por haber predicho a dónde hemos llegado [en tecnología de información]. Ocurre que principalmente se trata de personas que escriben ficción para vivir.”

Nick Donofrio, 2001 Turing Memorial Lecture

TEMAS DEL CAPÍTULO

- 3.1** De los componentes a las aplicaciones
- 3.2** Sistemas de cómputo y sus partes
- 3.3** Generaciones de progreso
- 3.4** Tecnologías de procesador y memoria
- 3.5** Periféricos, I/O y comunicaciones
- 3.6** Sistemas de software y aplicaciones

A la arquitectura de computadoras la impulsan los desarrollos en tecnología de cómputo y, a su vez, motiva e influye tales desarrollos. Este capítulo ofrece algunos antecedentes acerca del progreso pasado y de las tendencias actuales en tecnología de computación en la medida necesaria para apreciar adecuadamente los temas del resto del libro. Luego de observar el desarrollo de los sistemas de cómputo a través del tiempo, se revisa de manera breve la tecnología de los circuitos digitales integrados. Entre otras cosas, se muestra cómo la ley de Moore derivada experimentalmente predice con precisión las mejoras en el rendimiento y la densidad de los circuitos integrados durante las dos décadas pasadas y lo que esta tendencia significa para la arquitectura de computadoras. A esta presentación le sigue una discusión de la tecnología de procesadores, memoria, almacenamiento masivo, entrada/salida y comunicación. El capítulo concluye con un repaso a los sistemas de software y las aplicaciones.

3.1 De los componentes a las aplicaciones

Artefactos de ingeniería electrónica, mecánica y óptica se encuentran en todos los sistemas de cómputo modernos. Este texto se interesa principalmente por los aspectos de manipulación y control de datos electrónicos del diseño de computadoras. Se dan por sentadas las partes mecánicas (teclado, disco, impresora) y ópticas (display), no porque sean menos importantes que las partes electrónicas, sino porque el foco como parte fundamental de la arquitectura de computadoras se encuentra en esto último. La cadena que vincula las capacidades de los componentes electrónicos, en un lado de la figura 3.1 a los dominios de aplicación que buscan los usuarios finales en el otro lado involucra muchas subdisciplinas de la ingeniería de computación y las especialidades asociadas. En esta cadena, el diseñador o arquitecto de computadoras ocupa una posición central entre los diseñadores de hardware, quienes enfrentan nociones a nivel lógico y de circuito, y los diseñadores de software, quienes se ocupan de la programación del sistema y las aplicaciones. Desde luego, lo anterior no debería sorprender: si éste hubiese sido un libro acerca de arreglos florales, ¡el florista se habría mostrado como el centro del universo!

Conforme avance de derecha a izquierda en la figura 3.1, encontrará niveles crecientes de abstracción. El diseñador de circuitos tiene la visión de nivel más bajo y aborda los fenómenos físicos que hacen que el hardware de la computadora realice sus tareas. El diseñador lógico se enfrenta principalmente con modelos como compuertas o flip-flops, discutidos en los capítulos 1 y 2, y se apoya en herramientas de diseño para acomodar cualesquiera consideraciones de circuito que se puedan mostrar mediante la abstracción imperfecta. El arquitecto de computadoras requiere conocimientos acerca de la visión a nivel lógico, aunque trate principalmente con principios lógico digitales de nivel superior como sumadores y archivos de registro. También debe estar informado respecto de conflictos en el área del diseño de sistemas, cuya meta es proporcionar una capa de software que facilite la tarea del diseño y desarrollo de aplicaciones. En otras palabras, el diseñador de sistemas aborda el hardware bruto con componentes clave de software que proteja al usuario de los detalles de la operación de hardware, formatos de almacenamiento de archivos, mecanismos de protección, protocolos de comunicación, etc., ofrecidos en lugar de una interfaz con la máquina de fácil uso. Finalmente, el diseñador de aplicaciones, quien tiene la visión de nivel más alto, usa las facilidades ofrecidas por el hardware y el software de nivel inferior para divisar soluciones a problemas de aplicación que interesan a un usuario particular o una clase de usuarios.

La arquitectura de computadoras, cuyo nombre intenta reflejar su similitud con la arquitectura de edificios (figura 3.2), se ha descrito acertadamente como la interfaz entre hardware y software. Tradicionalmente, el lado de software se asocia con las “ciencias de la computación” y el lado del hardware

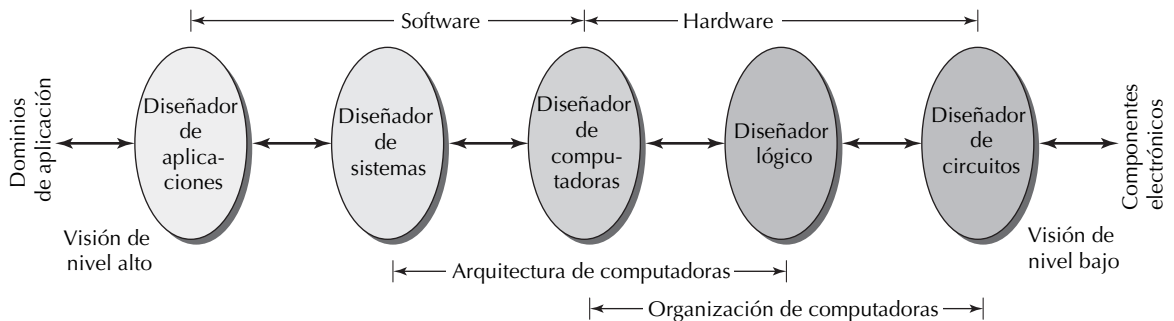


Figura 3.1 Subcampos o visiones en la ingeniería de sistemas de cómputo.

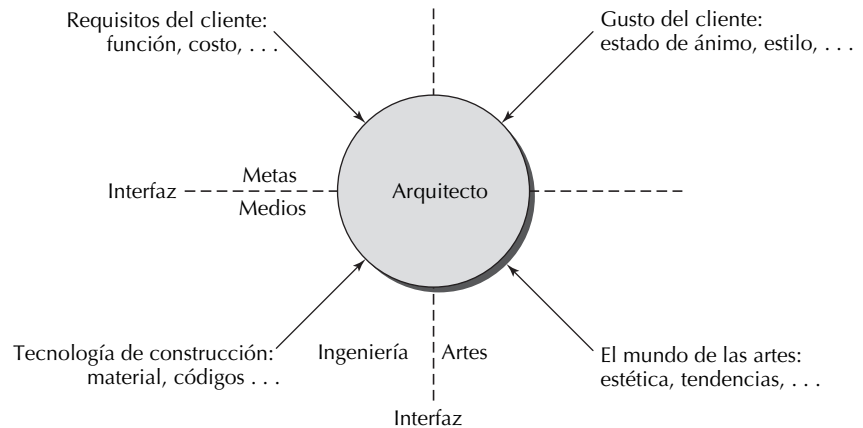


Figura 3.2 Como en la arquitectura de edificios, cuyo lugar en las interfaces ingeniería/arte y metas/medios ocupa el centro de este diagrama, un arquitecto de computadoras reconcilia muchas demandas conflictivas o competidoras.

con la “ingeniería de computación”. Esta dicotomía es un poco engañosa, pues existen muchas consideraciones científicas y de ingeniería en ambos lados. Asimismo, el desarrollo de software es una actividad de ingeniería; el proceso de diseñar un moderno sistema operativo o un sistema de gestión de base de datos sólo es superficialmente diferente del de diseñar un avión o un puente de suspensión. En este contexto, gran parte de las rutinas diarias de los diseñadores de hardware involucran software y programación (como en los dispositivos lógicos programables, los lenguajes de descripción de hardware y las herramientas de simulación de circuitos). Satisfacer metas de tiempo, costo y rendimiento, que se requieren en proyectos tanto de software como de hardware, son hitos de las actividades de ingeniería, como lo es la adhesión a los estándares de compatibilidad e interoperabilidad de los productos resultantes.

La arquitectura de computadoras no es sólo para los diseñadores y constructores de máquinas; los usuarios informados y efectivos en cada nivel se benefician de un firme entendimiento de las ideas fundamentales y de una comprensión de los conceptos más avanzados en este campo. Ciertas realizaciones clave, como el hecho de que un procesador 2x GHz no necesariamente es el doble de rápido que un modelo de x GHz (vea el capítulo 4), requieren un entrenamiento básico en arquitectura de computadoras. En cierta forma, usar una computadora es algo semejante a manejar un automóvil. Se puede hacer un buen trabajo con sólo aprender acerca de las interfaces del conductor en el automóvil y las reglas del camino. Sin embargo, para ser realmente un buen conductor, se debe tener cierto conocimiento respecto de cómo se hace andar a un automóvil y de cómo se interrelacionan los parámetros que afectan el rendimiento, la comodidad y la seguridad.

Puesto que al describir los sistemas de cómputo y sus partes se encuentran cantidades tanto extremadamente grandes como muy pequeñas (discos multigigabyte, elementos de circuito subnanómetro, etc.), se incluye la tabla 3.1, en la que se citan los prefijos prescritos por el sistema métrico de unidades y clarifica la convención con respecto a los prefijos que se usan en este libro para describir capacidad de memoria (kilobytes, gigabits, etc.). En este sentido, se incluyeron algunos múltiplos muy grandes y fracciones extremadamente pequeñas que no se han usado en conexión con la tecnología de computadoras. Conforme lea las secciones siguientes, al ritmo de progreso en esta área, comprenderá por qué esto último se puede volver relevante en un futuro no tan distante.

■ **TABLA 3.1** Símbolos y prefijos para múltiplos y fracciones de unidades.

Múltiplo	Símbolo	Prefijo	Múltiplo	Símbolo*	Prefijo*	Fracción	Símbolo	Prefijo
10 ³	k	kilo	2 ¹⁰	K o k _b	b-kilo	10 ⁻³	m	mili
10 ⁶	M	mega	2 ²⁰	M _b	b-mega	10 ⁻⁶	μ o u	micro
10 ⁹	G	giga	2 ³⁰	G _b	b-giga	10 ⁻⁹	n	nano
10 ¹²	T	tera	2 ⁴⁰	T _b	b-tera	10 ⁻¹²	p	pico
10 ¹⁵	P	peta	2 ⁵⁰	P _b	b-peta	10 ⁻¹⁵	f	femto
10 ¹⁸	E	exa	2 ⁶⁰	E _b	b-exa	10 ⁻¹⁸	a	atto
10 ²¹	Y	yotta	2 ⁷⁰	Y _b	b-yotta	10 ⁻²¹	y	yocto

* Nota: El símbolo K por lo general se usa para significar 2¹⁰ = 1 024. En virtud de que la misma convención no se puede aplicar a otros múltiplos cuyos símbolos ya son letras mayúsculas, se usa un subíndice b para denotar potencias comparables de 2. Cuando se especifica la capacidad de memoria, el subíndice b siempre se entiende y se puede quitar; es decir: 32 MB y 32 M_bB representan la misma cantidad de memoria. Los prefijos de potencia de 2 se leen como binario-kilo, binario-mega, etcétera.

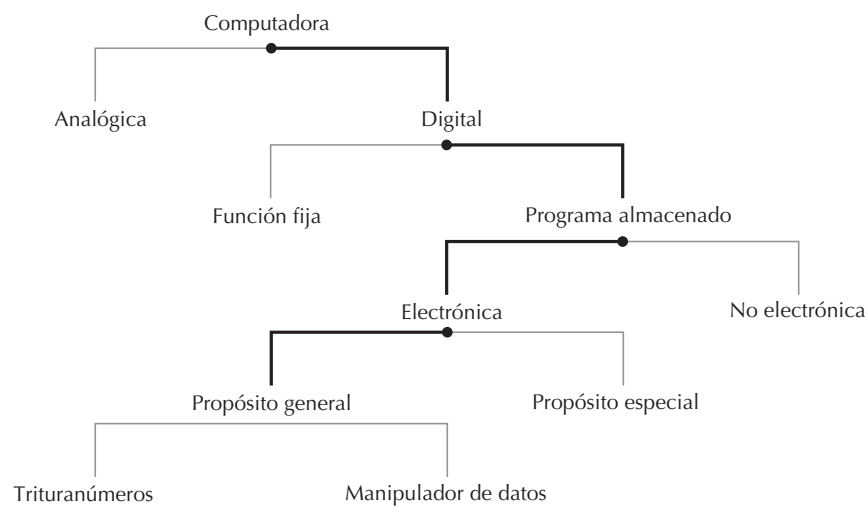


Figura 3.3 El espacio de los sistemas de cómputo: las líneas gruesas apuntan a lo que normalmente se entiende con el término “computadora”.

■ **3.2 Sistemas de cómputo y sus partes**

Las computadoras se pueden clasificar de acuerdo con su tamaño, potencia, precio, aplicaciones, tecnología y otros aspectos de su implementación o uso. Cuando en la actualidad se habla de computadoras, usualmente se hace alusión a un tipo de computadora cuyo nombre completo es “computadora digital de programa almacenado, electrónica y de propósito general” (figura 3.3). Tales computadoras solían diseñarse en versiones optimizadas para trituración de números, necesarias en cálculos numéricamente intensos y manipulación de datos, usualmente de aplicaciones de negocios. Las diferencias entre estas dos categorías de máquinas no han desaparecido en años recientes. Las computadoras analógicas, de función fija, no electrónicas y de propósito especial también se usan en gran cantidad, pero el enfoque de este libro para el estudio de la arquitectura de computadoras se encuentra en el tipo más común de computadora, que corresponde a la trayectoria que se destaca en la figura 3.3.

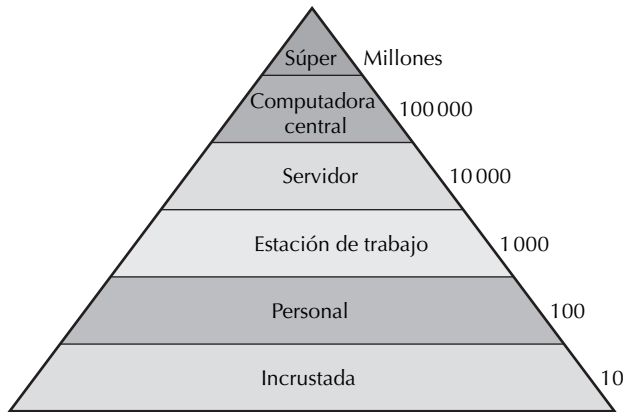


Figura 3.4 Clasificación de computadoras por potencia computacional y rango de precio (dólares).

Una categorización útil de computadoras se basa en su potencia computacional o rango de precio: desde los pequeños *microcontroladores embebidos* (básicamente, microprocesadores simples y de bajo costo) en la base de la escala, a través de las *computadoras personales*, *estaciones de trabajo* y *servidores* en la parte media, hasta las *computadoras centrales* (*mainframes*) y las *supercomputadoras* en la cima. La forma de pirámide de la figura 3.4 es útil para aclarar que las computadoras más cercanas a la base son más numerosas y también representan mayor inversión en su costo agregado. Conforme las computadoras han logrado mejores razones rendimiento/costo, la demanda del usuario por potencia computacional ha crecido a un ritmo incluso más rápido, de modo que las supercomputadoras en la cima tienden a costar entre diez y 30 millones de dólares. De igual modo, el costo de una computadora personal de potencia modesta oscila entre los mil y tres mil dólares. Estas observaciones, junto con la naturaleza enormemente competitiva del mercado de computadoras, implican que el costo de un sistema de cómputo es un indicador sorprendentemente preciso de sus capacidades y potencia computacional, al menos más exacto que cualquier otro indicador numérico individual. Sin embargo, observe que los costos que se muestran en la figura 3.4 se expresan en números redondos fáciles de recordar.

Las computadoras embebidas se usan en aparatos electrodomésticos, automóviles, teléfonos, cámaras y sistemas de entretenimiento, y en muchos artilugios modernos. El tipo de computadora varía con las tareas de procesamiento pretendidas. Los electrodomésticos tienden a contener *microcontroladores* muy simples capaces de dar seguimiento a información de estado a partir de sensores, convertir entre señales analógicas y digitales, medir intervalos de tiempo y activar o deshabilitar actuadores mediante la postulación de señales de habilitación e inhibición. Las aplicaciones automotrices más actuales son similares, excepto que requieren microcontroladores de tipos más resistentes. No obstante, con más frecuencia se espera que las computadoras instaladas en los automóviles realicen funciones de control más que simples (figura 3.5), dada la tendencia hacia mayor uso de información y artículos de entretenimiento. Los teléfonos, cámaras digitales y sistemas de entretenimiento de audio/video requieren muchas funciones de procesamiento de señales sobre datos multimedia. Por esta razón, tienden a usar un chip *procesador de señal digital* (DSP, por sus siglas en inglés) en lugar de un microcontrolador.

Las computadoras personales son igualmente variadas en sus potenciales computacionales y pretensiones de uso. Se les identifica en dos categorías principales: *portátiles* y *de escritorio*. Las primeras se conocen como *laptops* o *notebooks*, dentro de las cuales hay *subnotebooks* y las *pocket PC* constituyen modelos más pequeños y más limitados. También están disponibles las versiones *tablet*, cuya intención es sustituir cuadernos y plumas. Una *computadora de escritorio* puede tener su CPU y periféricos en una unidad que se duplica si el monitor lo soporta o se encuentra en una “torre” separada (figura 3.6) que ofrece más espacio para expansión y también se puede esconder de la vista.

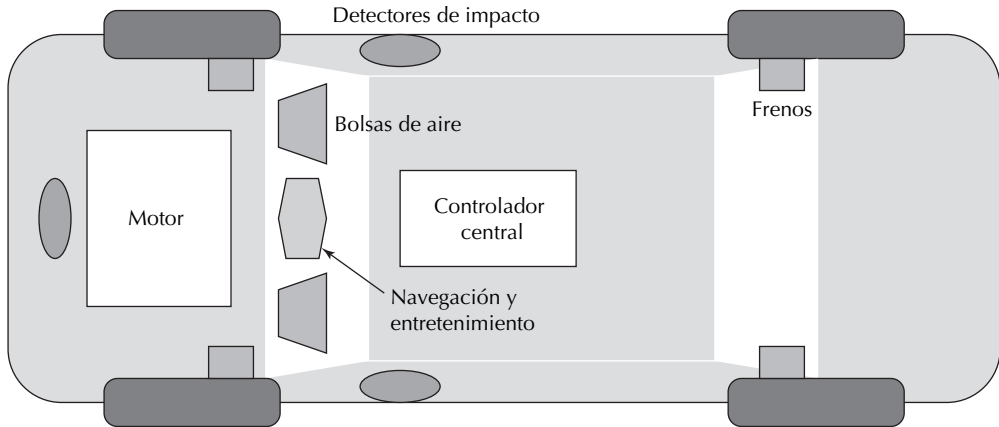


Figura 3.5 Las computadoras embebidas son omnipresentes, aunque invisibles. Se encuentran en los automóviles, electrodomésticos y muchos otros lugares.

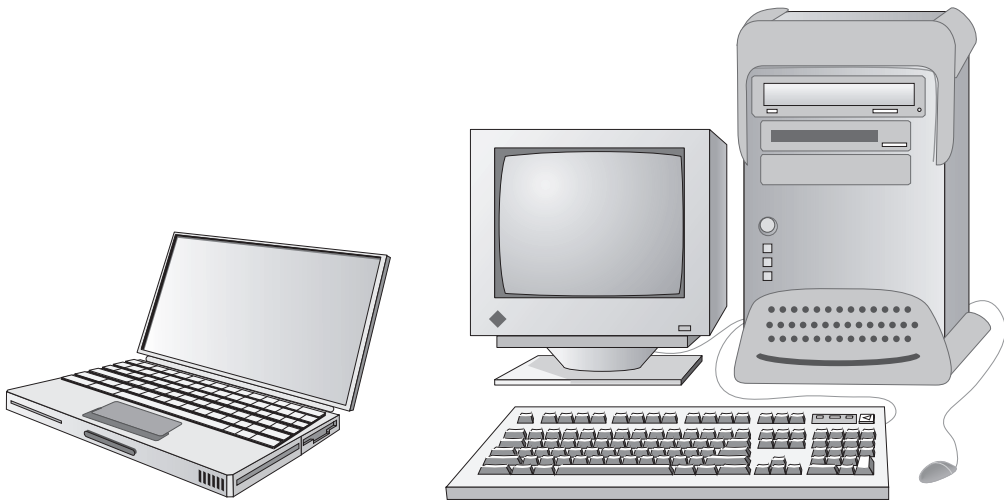


Figura 3.6 Las notebook, una clase común de computadoras portátiles, son mucho más pequeñas que las de escritorio, pero ofrecen sustancialmente las mismas capacidades. ¿Cuáles son las principales razones de la diferencia de tamaño?

Dado que los displays planos ahora son bastante accesibles, las versiones de escritorio cada vez más vienen con pantallas planas que ofrecen los beneficios de imágenes más claras, consumo de potencia más bajo, y menos disipación de calor (por tanto, menor costo de acondicionamiento de aire). Una *estación de trabajo* (*workstation*) constituye, básicamente, una computadora de escritorio con más memoria, mayores capacidades I/O y de comunicaciones, pantalla de despliegue más grande y sistema y software de aplicaciones más avanzadas.

Mientras que una computadora personal o estación de trabajo por lo general se asocian con un solo usuario, los *servidores* y *computadoras centrales* son computadoras de grupos de trabajo, o departamentales, o empresariales. En el perfil bajo, un servidor es muy parecido a una estación de trabajo y se distingue de ella por software/soporte más extenso, memoria principal más larga, almacenamiento secundario más grande, capacidades I/O mayores, mayor rapidez/capacidad de comunicación y mayor

confiabilidad. Su capacidad de trabajo y confiabilidad son factores particularmente importantes, dadas las severas y recurrentes consecuencias técnicas y financieras de las caídas imprevistas. Es común usar servidores múltiples para satisfacer los requisitos de capacidad, ancho de banda o disponibilidad. En el extremo, este enfoque conduce al uso de *granjas servidoras*. Un servidor de perfil alto es, básicamente, una versión más pequeña o menos costosa de una computadora central, en décadas pasadas se habría llamado *minicomputadora*. Una computadora central, con todos sus dispositivos periféricos y equipo de soporte, puede llenar una habitación o un área grande.

Las supercomputadoras representan los productos glamorosos de la industria de las computadoras. Representan una pequeña fracción de la cantidad monetaria total de los productos embarcados e incluso una fracción todavía más pequeña del número de instalaciones de computadoras. No obstante, estas máquinas y los enormemente desafiantes problemas computacionales que motivan su diseño y desarrollo siempre se llevan los encabezados. Parte de la fascinación con las supercomputadoras resulta de saber que muchos de los avances introducidos en tales máquinas con frecuencia encuentran su camino pocos años después hacia las estaciones de trabajo y computadoras de escritorio; de modo que, en cierto sentido, ofrecen una ventana del futuro. Una supercomputadora se ha definido, medio en broma, como “cualquier máquina que todavía se encuentra en la mesa de dibujo” y “cualquier máquina que cuesta 30 mil dólares”. La potencia computacional de las más grandes computadoras disponibles ha pasado de millones de instrucciones u operaciones de punto flotante por segundo (MIPS, por sus siglas en inglés, FLOPS) en la década de 1970, a GIPS/GFLOPS a mediados del decenio de 1980 y a TIPS/TFLOPS a finales del siglo xx. Las máquinas que ahora se encuentran en la mesa de dibujo se dirigen a los PIPS/PFLOPS. Además de los cálculos numéricamente intensos que siempre se asocian con las supercomputadoras, tales máquinas se usan cada vez más para almacenamiento de datos y procesamiento de transacción de alto volumen.

Sin importar su tamaño, rango de precio o dominio de aplicación, una computadora digital se compone de ciertas partes clave, que se muestran en la figura 3.7. Este diagrama tiene la intención de reflejar todas las diferentes visiones que se han avanzado; desde la visión tripartita (CPU, memoria, I/O), pasando por las de cuatro (procesador, memoria, entrada, salida) y cinco (trayectoria de datos, control, memoria, entrada, salida) partes que se ven en la mayoría de los libros de texto, hasta la visión de seis partes preferida por este texto, que incluye una mención explícita del componente de vinculación (*link*). Observe que cualquiera de estas versiones, incluida la del texto, representan una visión simplificada que aglomera funcionalidades que, de hecho, se pueden distribuir a través del sistema. Ciertamente, esto es verdadero para el control: con frecuencia, la memoria tiene un controlador separado, así

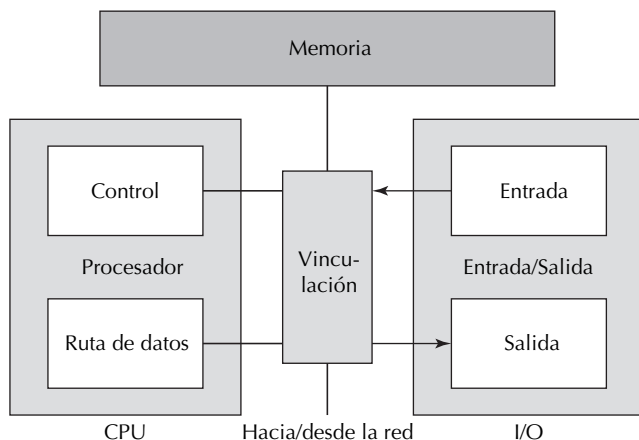


Figura 3.7 Se observan las (tres, cuatro, cinco o) seis unidades principales de una computadora digital. Usualmente, la unidad de vinculación (un simple bus o una red más elaborada) no se incluía de manera explícita en tales diagramas.

como el subsistema entrada/salida, los dispositivos I/O individuales, la interfaz de red, etc. De igual modo, existen capacidades de procesamiento por todas partes: en el subsistema I/O, dentro de varios dispositivos (por ejemplo, impresora), en la interfaz de red, etc. Sin embargo, ¡tenga en mente que no aprendemos de aviones al usar al principio un *Concorde* o un sofisticado jet militar!

■ 3.3 Generaciones de progreso

La mayoría de las cronologías de tecnología de computadoras distingue un número de *generaciones de computadoras*, y cada una comienza con un gran adelanto en la tecnología de componentes. Usualmente se especifican cuatro generaciones que coinciden con el uso de tubos de vacío, transistores, circuitos integrados de pequeña a media escala (CI SSI/MSI, por sus siglas en inglés) e integración de grande a muy grande escala (LSI/VLSI, por sus siglas en inglés). A esto último agregamos toda la historia previa de la computación como generación 0 y los dramáticos avances de finales de la década de 1990 como generación 5 (tabla 3.2). Observe que las entradas en la tabla 3.2 están un tanto sobresimplificadas; la intención es mostrar una cuadro de los avances y tendencias en aspectos amplios, más que una tabla confusa que mencione todos los avances e innovaciones tecnológicos.

La preocupación con la computación y los artilugios mecánicos para facilitarla comenzaron muy temprano en la historia de la civilización. Desde luego, es bien conocida la introducción del ábaco por los chinos. Los primeros auxiliares sofisticados para la computación parecen haberse desarrollado en la antigua Grecia para cálculos astronómicos. Para propósitos prácticos, las raíces de la computación digital moderna se remontan hasta las calculadoras mecánicas diseñadas y construidas por los matemáticos del siglo xvii¹ en la década de 1820, y el motor analítico programable más o menos una década después, Charles Babbage se autoestableció como el padre de la computación digital, como se le conoce a esta última hoy día. Aun cuando la tecnología del siglo xix no permitió una implementación completa de las ideas de Babbage, los historiadores concuerdan en que él tuvo realizaciones operativas de muchos conceptos clave en la computación, incluidos la programación, el conjunto de instrucciones, los códigos de operación y los ciclos de programa. La aparición de relevadores electromecánicos confiables en la década de 1940 propició muchas de estas ideas.

■ TABLA 3.2 Las cinco generaciones de computadoras digitales y sus antecesores.

Generación (cuándo comenzó)	Tecnología de procesador	Principales innovaciones de memoria	Dispositivos I/O introducidos	Apariencia y gusto dominante
0 (siglo xvii)	(Electro)mecánico	Rueda, tarjeta	Palanca, dial, tarjeta perforada	Equipo de fábrica
1 (1950)	Tubo de vacío	Tambor magnético	Cinta de papel, cinta magnética	Gabinete de alimentación tamaño salón
2 (1960)	Transistor	Núcleo magnético	Tambor, impresora, terminal de texto	Computadora central tamaño habitación
3 (1970)	SSI/MSI	Chip RAM/ROM	Disco, teclado, monitor de video	Minitamaño escritorio
4 (1980)	LSI/VLSI	SRAM, DRAM	Red, CD, ratón, sonido	Desktop/laptop micro
5 (1990)	ULSI/GSI/WSI sistema en chip	SDRAM, flash	Sensor, actuador, punto/click, DVD	Invisible, embebida

¹ Schilcard, Pascal y Leibniz, entre otros. Al inventar su motor de diferencia para el cálculo de las tablas matemáticas.

Con frecuencia se cita a la ENIAC, construida en 1945 bajo la supervisión de John Mauchly y J. Presper Eckert en la Universidad de Pennsylvania, como la primera computadora electrónica, aunque ahora se sabe de muchos otros esfuerzos concurrentes o anteriores; por ejemplo, los de John Atanasoff en la Universidad Estatal de Iowa y Konrad Zuse en Alemania. La ENIAC pesaba 30 toneladas, ocupaba casi 1 500 m² de espacio de piso, usaba 18 000 tubos de vacío y consumía 140 kW de electricidad. Podía realizar alrededor de cinco mil sumas por segundo. La noción de calcular con un programa almacenado se divisó y perfeccionó en grupos encabezados por John von Neumann, en Estados Unidos, y Maurice Wilkes, en Inglaterra, ello condujo a máquinas funcionales a finales de la década de 1940 y a computadoras digitales comerciales para aplicaciones científicas y empresariales a principios del decenio de 1950. Estas computadoras de primera generación, digitales, con programa almacenado y de propósito general estaban adaptadas para aplicaciones científicas o empresariales, distinción que prácticamente ha desaparecido. Los ejemplos de máquinas de primera generación incluyen las series 1100 de UNIVAC y 700 de IBM.

El surgimiento de la segunda generación se asocia con el cambio de los tubos de vacío a transistores mucho más pequeños, más baratos y más confiables. Sin embargo, igualmente importantes, si no es que más, son los desarrollos en tecnología de almacenamiento junto con la introducción de lenguajes de programación de alto nivel y software de sistema. NCR y RCA fueron los pioneros de los productos computacionales de segunda generación, fueron seguidos por la serie 7000 de máquinas de IBM y, más tarde, por la PDP-1 de Digital Equipment Corporation (DEC). Estas computadoras comenzaron a parecer más máquinas de oficina que equipos de fábrica. Este hecho, además de la facilidad de uso proporcionada por software más sofisticado, condujo a la proliferación de computación científica y a aplicaciones de procesamiento de datos empresariales.

La capacidad para integrar muchos transistores y otros elementos, hasta entonces contruidos como *componentes discretos*, en un solo circuito resolvió muchos problemas que comenzaban a volverse bastante serios conforme crecía la complejidad de las computadoras a decenas de miles de transistores y más. Los *circuitos integrados* no sólo trajeron la tercera generación de computadoras, sino que también impulsaron una *revolución microelectrónica* abarcadora que continúa dando forma a la sociedad basada en información. Acaso el producto computacional de tercera generación más exitoso e influyente, que también ayudó a poner atención en la arquitectura de computadoras como algo distinto de las máquinas particulares o tecnologías de implementación, es la familia IBM System 360 de máquinas compatibles. Esta serie comenzó con máquinas de perfil bajo relativamente baratas para negocios pequeños y se extendió a multimillonarias supercomputadoras muy grandes que usaban las más recientes innovaciones tecnológicas y algorítmicas para lograr el máximo rendimiento, todo ello con base en la misma arquitectura global y conjunto de instrucciones. Otra máquina influyente en esta generación, la PDP-11 de DEC, trajo consigo la época de minicomputadoras costeables capaces de operar en la esquina de una oficina o laboratorio, en lugar de requerir un gran cuarto de cómputo con aire acondicionado.

Conforme los circuitos integrados se volvieron más grandes y densos, finalmente fue posible, a comienzos de la década de 1970, colocar un procesador completo, aunque muy simple, en un solo chip CI. Lo anterior, además de incrementos fenomenales en densidad de memoria que permitieron que la memoria principal completa residiera en un puñado de chips, condujo a la popularización de microcomputadoras de bajo costo. La Apple Computer Corporation era el líder indiscutible en esta área, pero no fue sino hasta que IBM introdujo su PC con arquitectura abierta (lo que significa que componentes, periféricos y software de diferentes empresas podían coexistir en una sola máquina) que la revolución de las PC se desarrolló notablemente. A la fecha, el término PC es sinónimo de IBM y de microcomputadoras compatibles con IBM. Las computadoras más grandes también se beneficiaron de los avances en tecnología IC; con cuatro generaciones de computadoras de escritorio que ofrecen las capacidades de las supercomputadoras desde las primeras generaciones, las nuevas máquinas de alto rendimiento continúan presionando hacia adelante el avance de la computación científica y de las aplicaciones comerciales de datos intensivos.

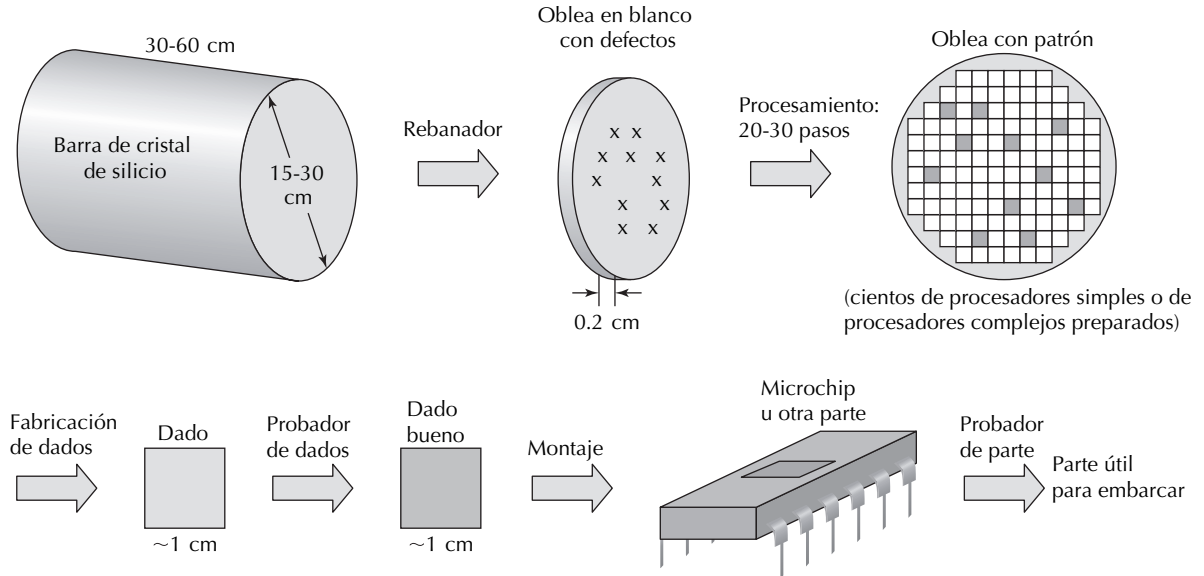


Figura 3.8 Proceso de fabricación de un circuito integrado.

No hay acuerdo general acerca de si ya se pasó de la cuarta generación y, si fuera así, cuándo ocurrió la transición a la quinta generación. En la década de 1990 se atestiguaron mejoras dramáticas no sólo en tecnología de IC sino también en comunicaciones. Si la llegada de las PC de bolsillo, las computadoras de escritorio GFLOPS, el acceso inalámbrico a Internet, los chips con gigabits de memoria y los discos multigigabytes, apenas más grandes que un reloj de pulsera, son inadecuados para señalar la alborada de una nueva era, es difícil imaginar cuál sería. La parte IC de estos avances se describe como integración a escala ultralarga, gran escala o a escala pequeña. Ahora es posible colocar un sistema completo en un solo chip IC, lo que conduce a rapidez, compactación y economía de electricidad sin precedentes. En la actualidad, las computadoras se visualizan principalmente como componentes dentro de otros sistemas en lugar de como sistemas costosos por su propio derecho.

Un breve vistazo al proceso de fabricación de circuitos integrados (figura 3.8) es útil para comprender algunas de las dificultades actuales, así como los retos a superar, para el crecimiento continuo en las capacidades de las computadoras en generaciones futuras. En esencia, los circuitos integrados se imprimen en dados de silicio con el uso de un complejo proceso químico en muchos pasos para depositar capas (aislamiento, conductor, etc.), eliminar las partes innecesarias de cada capa y proceder con la capa siguiente. Se pueden formar cientos de dados a partir de una sola oblea en blanco rebanada de una barra de cristal. Puesto que la barra y el proceso de depósito (que involucra características minúsculas en capas extremadamente delgadas) son imperfectos, algunos de los dados así formados no se desempeñarán como se espera y entonces se deben desechar. Durante el proceso de montaje surgen otras imperfecciones, ello conduce a más partes descartadas. A la razón de las partes utilizables obtenidas al final de este proceso, así como al número de dados con los que se comenzó, se le denomina *producción* del proceso de fabricación.

Muchos factores afectan la producción. Un factor clave es la complejidad del dado en términos del área que ocupa y lo intrincado de su diseño. Otro es la densidad de defectos en la oblea. La figura 3.9 muestra que una distribución de 11 defectos pequeños en la superficie de la oblea pueda conducir a 11 dados defectuosos entre 120 (producción = $109/120 \cong 91\%$), mientras que los mismos defectos harían inutilizables 11 de 26 dados más grandes (producción = $15/26 \cong 58\%$). Ésta es la única contribución

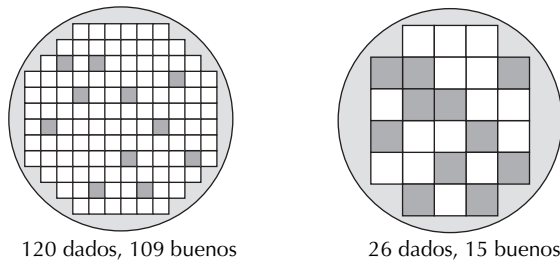


Figura 3.9 Visualización de la disminución dramática en la producción con dados más grandes.

de los defectos de oblea; la situación empeorará por los defectos que surjan de la estructura más compleja y los patrones de interconexión en los dados más grandes.

Si se comienza con la definición

Producción dados = (número dados buenos)/(número total dados)

se llega a lo siguiente para el costo de un dado, exclusivo de costos de poner a prueba y de empaquetado:

$$\begin{aligned}\text{Costo dado} &= (\text{costo oblea})/(\text{número total dados} \times \text{producción dados}) \\ &= (\text{costo oblea}) \times (\text{área dado}/\text{área oblea})/(\text{producción dados})\end{aligned}$$

Las únicas variables en la ecuación precedente son área de dado y producción. Como consecuencia de que la producción es una función decreciente del área de dado y de la densidad de defectos, el costo por dado constituye una función superlineal del área de dado, ello significa que duplicar el área de dado para acomodar mayor funcionalidad en un chip más que duplicará el costo de la parte terminada. En concreto, note que los estudios experimentales demuestran que la producción de dados es

$$\text{Producción dados} = \text{producción obleas} \times [1 + (\text{densidad de defectos} \times \text{área dado})/a]^{-a}$$

donde la producción de obleas representa las que son completamente inutilizables y el parámetro a se estima que varía de 3 a 4 para procesos CMOS modernos.

Ejemplo 3.1: Efecto del tamaño de dado en el costo Suponga que los dados de la figura 3.9 miden 1×1 y 2×2 cm² e ignore el patrón de defecto que se muestra. Si se supone una densidad de defecto de $0.8/\text{cm}^2$, ¿cuánto más costoso será el dado de 2×2 que el dado de 1×1 ?

Solución: Sea w la producción de obleas. A partir de la fórmula de producción de dado, se obtiene una producción de $0.492w$ y $0.113w$ para los dados de 1×1 y 2×2 , respectivamente, si se supone $a = 3$. Al poner estos valores en la fórmula para costo de dado, se encuentra que el dado 2×2 cuesta $(120/26) \times (0.492/0.113) = 20.1$ veces el dado de 1×1 ; esto último representa un factor de $120/26 = 4.62$ costo mayor atribuible al número más pequeño de dados en una oblea y un factor de $0.492/0.113 = 4.35$ debido al defecto de producción. Con $a = 4$, la razón supone el valor un poco más grande $(120/26) \times (0.482/0.095) = 23.4$.

■ 3.4 Tecnologías de procesador y memoria

Como resultado de los avances en electrónica, el procesador y la memoria han mejorado de manera sorprendente. La necesidad de procesadores más rápidos y memorias más amplias alimenta el crecimiento fenomenal de la industria de semiconductores. Un factor clave en estas mejoras ha sido el

crecimiento incansable en el número de dispositivos que se pueden poner en un solo chip. Parte de este crecimiento resultó de la habilidad para diseñar y fabricar económicamente dados más grandes; pero, principalmente, la densidad creciente de los dispositivos (por unidad de área de dado) es el factor clave. El aumento exponencial en el número de dispositivos en un chip a lo largo de los años se denomina *ley de Moore*, que predice un aumento anual de 60% ($\times 1.6$ por año $\cong \times 2$ cada 18 meses $\cong \times 10$ cada cinco años). Esta predicción ha resultado ser tan precisa que un plan a largo plazo, conocido como mapa de la industria de semiconductores, se basa en ella. Por ejemplo, de acuerdo con este mapa se sabe con certeza que en 2010 los chips con miles de millones de transistores serán técnica y económicamente factibles (ya están disponibles chips de memoria de ese tamaño).

En la actualidad, los procesadores más conocidos son los de la familia de chips Pentium, de Intel, y los productos compatibles que ofrecen AMD y otros fabricantes. El procesador Pentium de 32 bits tiene sus raíces en el chip 8086 de 16 bits y su acompañante coprocesador de punto flotante 8087 introducido por Intel a finales de la década de 1970. Conforme se sintió la necesidad de máquinas de 32 bits, que entre otras cosas pueden manipular direcciones de memoria más amplias, Intel introdujo los procesadores 80386 y 80486 antes de mudarse hacia Pentium y sus modelos mejorados que se identifican con los sufijos II, III y IV (o 4). Las versiones más recientes de estos chips no sólo contienen la unidad de punto flotante en el mismo chip, también tienen memorias caché en el chip. Cada modelo en esta secuencia de productos introduce mejoras y extensiones al modelo previo, pero la arquitectura del conjunto de instrucciones central no se altera. Al momento de escribir este libro se introdujo la arquitectura Itanium de 64 bits para aumentar aún más el espacio direccionable y la potencia computacional de la serie Pentium. Aunque Itanium no es una simple extensión de Pentium, está diseñado para correr programas escritos para lo más avanzado de estos sistemas. Power PC, hecha por IBM y Motorola, que deriva su fama de la incorporación en computadoras Apple, es un ejemplo de modernos procesadores RISC (vea el capítulo 8). Otros ejemplos de esta categoría incluyen productos de MIPS y el procesador DEC/Compaq Alpha.

Como resultado de circuitos más densos, y, por ende, más rápidos, junto con mejoras arquitectónicas, el rendimiento del procesador creció exponencialmente. La ley de Moore que predice un factor de mejora de 1.6 por año en la densidad de componentes, también es aplicable a la tendencia en el rendimiento de procesador medido en las instrucciones por segundo (IPS) ejecutadas. La figura 3.10 muestra esta tendencia junto con datos puntuales para algunos procesadores clave (Intel Pentium y sus predecesores 80x86, la serie 6800 de Motorola y MIPS R10000). La ley de Moore, cuando se aplica a chips de memoria, predice mejoramiento de capacidad por un factor de 4 cada tres años (la capacidad del chip de memoria casi siempre es una potencia par de 2; de ahí lo apropiado de la formulación del factor de 4). La figura 3.10 muestra la tendencia de chip de memoria desde 1980 y su proyección durante los siguientes años. Con el chip gigabit (128 MB), la necesidad de memoria de una PC típica encaja en uno o un puñado de chips. El siguiente desafío a corto plazo es poner procesador y memoria en el mismo dispositivo, así como habilitar una computadora personal compuesta de un solo chip.

Los chips de procesador y memoria deben conectarse unos a otros y a las otras partes de la computadora que se muestra en la figura 3.7. Para este propósito se usan varios esquemas de empaquetado, dependiendo del tipo de computadora y los objetivos costo/rendimiento. La forma más común de empaquetado se bosqueja en la figura 3.11a. Los chips de memoria se montan en pequeñas placas de circuito impreso (PC, por sus siglas en inglés) conocidas como *tarjetas hijas*. Luego una o más de éstas, usualmente con una sola hilera de pines de conexión (módulos individuales de memoria en línea o SIMM = *single in-line memory modules*), se montan en la *tarjeta madre* (*motherboard*) que sostiene al procesador, el bus de sistema, varias interfases y una diversidad de conectores. Todos los circuitos para una computadora pequeña pueden encajar en una sola tarjeta madre, mientras que las máquinas más grandes pueden requerir muchas de tales placas montadas en un *chasis* o *jaula de tarjetas* e interconectarse mediante una *tarjeta base*. La tarjeta madre sola, o el chasis que sostiene múltiples placas,

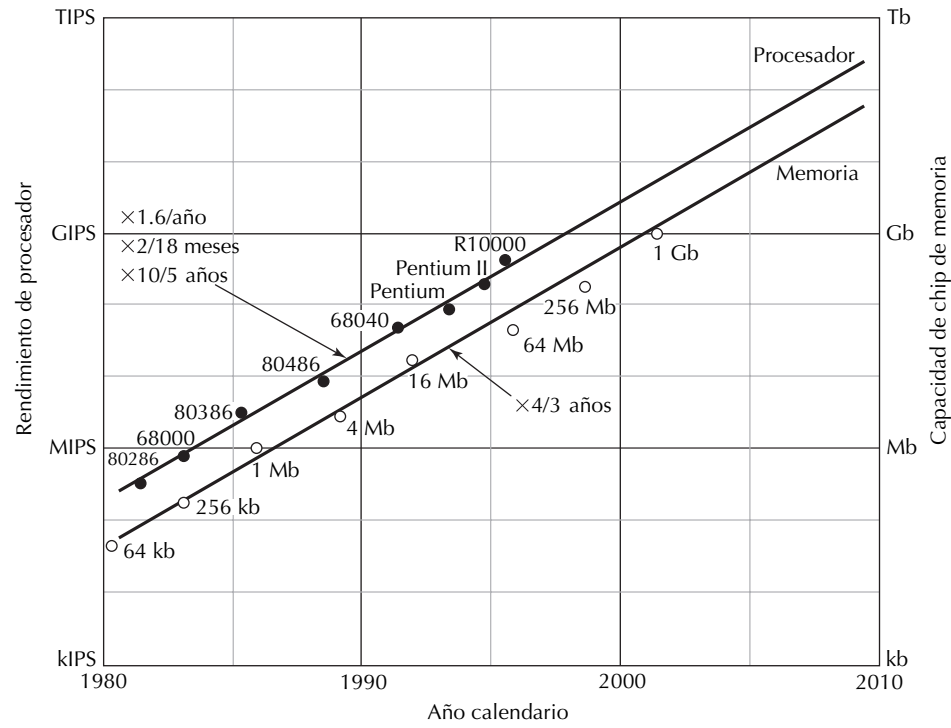


Figura 3.10 Tendencias en rendimiento de procesador y capacidad de chip de memoria DRAM (ley de Moore).

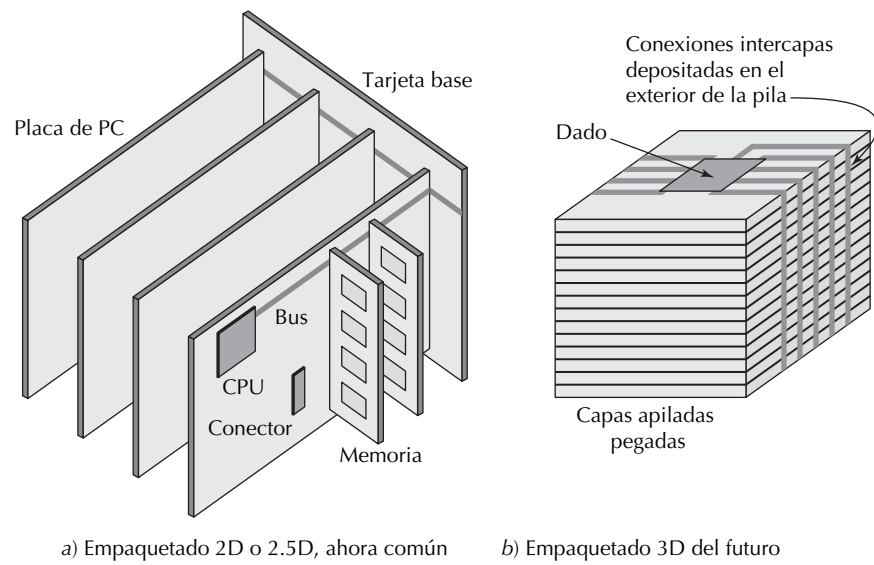


Figura 3.11 Empaquetado de procesador, memoria y otros componentes.

se empaqueta con dispositivos periféricos, fuentes de poder, ventilador de enfriamiento y otros componentes requeridos en una caja o gabinete, ello deja suficiente espacio interno para expansión futura.

De hecho, todos los más o menos 10^{18} transistores que se estima se han incorporado en los circuitos integrados hasta la fecha se construyeron a “nivel del suelo”, directamente en la superficie de los cristales de silicio. Tal como uno puede aumentar la densidad de población en una ciudad al echar mano de edificios de muchos pisos y gran altura, es posible encajar más transistores en un volumen específico al usar empaquetado 3D [Lee02]. La investigación es continua en la tecnología de empaquetado 3D y han comenzado a surgir productos que incorporan estas tecnologías. Un esquema promisorio bajo consideración (figura 3.11b) permite la vinculación directa de los componentes a través de conectores depositados en el exterior de un cubo 3D formado por apilamiento de circuitos integrados 2D, así como de la remoción de gran parte del volumen y costo de los métodos actuales.

Las mejoras dramáticas en el rendimiento de procesador y la capacidad de los chips de memoria se acompañan con reducciones de costo igualmente significativas. Tanto la potencia de computación como la capacidad de memoria por costo unitario (MIPS/\$ o MFLOPS/\$, MB/\$) aumentaron exponencialmente durante las dos décadas pasadas y se espera que continúe en este curso para el futuro cercano. Hasta hace casi una década solía afirmarse, en broma, que pronto se tendrían componentes de costo cero o negativo. Hace poco supieron los autores que, a la vuelta del siglo XXI, el hardware de computadora se estaba alejando de la anticipación de la rentabilidad para hacerse de servicios y productos de software.

Pararon el propósito de concretar el significado de dichas mejoras en el rendimiento de las computadoras y las acompañantes reducciones de costo, a veces se usa una interesante comparación. Se afirma que si la industria de la aviación hubiese avanzado a la misma tasa que la industria de computadoras, viajar de Estados Unidos a Europa ahora tomaría unos cuantos segundos y costaría pocos centavos. Una analogía similar, aplicada a la industria automotriz, conduciría a la expectativa de ser capaces de comprar un auto de lujo que viajara tan rápido como un jet y corriera para siempre con un solo tanque de gasolina, por el mismo precio que una taza de café. Desafortunadamente, la confiabilidad del software de aplicaciones y sistemas no ha mejorado a la misma tasa, esto último conduce a la afirmación contraria de que si la industria de computadoras se hubiese desarrollado en la misma forma que la industria de transportes, ¡el sistema operativo Windows se caería no más de una vez en un siglo!

■ 3.5 Periféricos, I/O y comunicaciones

El estado de la computación no habría sido posible sólo con las mejoras en el rendimiento de procesadores y densidad de memoria discutidos en la sección 3.4. El fenomenal progreso en tecnologías de entrada/salida, desde las impresoras y escáner a las unidades de almacenamiento masivo e interfaces de comunicación, han sido importantes. Los discos duros de 100 dólares de hoy, que encajan fácilmente en la más delgada de las computadoras notebook, pueden almacenar tantos datos como cabrían en una sala de una casa llena de gabinetes empacada con docenas de discos muy costosos de la década de 1970. Los dispositivos de entrada/salida (I/O) se discuten con mayor detalle en el capítulo 21. Aquí se presenta un breve panorama de los tipos de dispositivo I/O y sus capacidades con la meta de completar el cuadro de amplias pinceladas de la moderna tecnología de computadoras. En la tabla 3.3 se mencionan las categorías principales de dispositivos I/O. Observe que no se mencionan la tarjeta perforada y los lectores de cinta de papel, terminales de impresión, tambores magnéticos y otros dispositivos que no se usan actualmente.

Los dispositivos de entrada se pueden categorizar en muchas formas. La tabla 3.3 usa el tipo de datos de entrada como la característica principal. Los tipos de datos de entrada incluyen símbolos de un alfabeto finito, información posicional, verificación de identidad, información sensorial, señales de audio, imágenes fijas y video. Dentro de cada clase se ofrecen ejemplos principales y ejemplos adicionales,

■ **TABLA 3.3** Algunos dispositivos de entrada, salida e I/O de dos vías.

Tipo entrada	Ejemplos principales	Otros ejemplos	Tasa datos (b/s)	Usos principales
Símbolo	Teclado, keypad	Nota musical, OCR	10	Ubicuo
Posición	Ratón, <i>touchpad</i>	Palanca, rueda, guante	100	Ubicuo
Identidad	Lector de código de barras	Insignia, huella digital	100	Ventas, seguridad
Sensorial	Toque, movimiento, luz	Esencia, señal cerebral	100	Control, seguridad
Audio	Microfono	Teléfono, radio, cinta	1 000	Ubicuo
Imagen	Escáner, cámara	Tableta gráfica	1 000 millones	Fotografía, publicidad
Video	Cámara de video, DVD	VCR, TV cable	1 000 miles de millones	Entretimiento
Tipo de salida	Ejemplos principales	Otros ejemplos	Tasa datos (b/s)	Usos principales
Símbolo	Segmentos de línea LCD	LED, luz de estatus	10	Motor caminador
Posición	Motor caminador	Movimiento robótico	100	Ubicuo
Advertencia	Zumbador, campana, sirena	Luz destellante	Unos cuantos	Salvaguarda, seguridad
Sensorial	Texto Braille	Esencia, estímulo cerebral	100	Auxilio personal
Audio	Bocina, audiocinta	Sintetizador de voz	1 000	Ubicuo
Imagen	Monitor, impresora	<i>Plotter</i> , microfilm	1 000	Ubicuo
Video	Monitor, pantalla TV	Grabadora película/video	1 000 miles de millones	Entretimiento
I/O de dos vías	Ejemplos principales	Otros ejemplos	Tasa datos (b/s)	Usos principales
Almacenamiento masivo	Disco duro/compacto	<i>Floppy</i> , cinta, archivo	Millones	Ubicuo
Red	Módem, fax, LAN	Cable, DSL, ATM	1 000 miles de millones	Ubicuo

junto con tasas de datos usuales y dominios de aplicación principales. Los dispositivos de entrada lenta son aquellos que no producen muchos datos y, por ende, no necesitan una gran cantidad de potencia de cómputo para ser atendidos. Por ejemplo, la tasa de entrada pico desde un teclado está limitada a cuán rápido pueden escribir los humanos. Si se suponen 100 palabras (500 bytes) por minuto, se obtiene una tasa de datos de alrededor de 67 b/s. Un procesador moderno podría manipular datos de entrada provenientes de millones de teclados, para el caso de que se requiriera. En el otro extremo, la entrada de video de alta calidad puede requerir la captura de millones de pixeles por cuadro, cada uno codificado en 24 bits (ocho bits por color primario), por decir, a una tasa de 100 cuadros por segundo. Esto último significa una tasa de datos de miles de millones de bits por segundo y desafía la potencia de las computadoras más rápidas disponibles en la actualidad.

Los dispositivos de salida se categorizan de igual modo en la tabla 3.3, pero la categoría en la hilera “identidad” se sustituye con la de “advertencia”. Teóricamente, la activación de una alarma requiere un solo bit y representa la tasa I/O más lenta posible. De nuevo, en el extremo alto, la salida de video en tiempo real puede requerir una tasa de datos de miles de millones de bits por segundo. Una impresora de alta rapidez, que imprima docenas de páginas a color por minuto, es poco menos demandante. Tanto para video como para imágenes fijas, la tasa de datos se puede reducir mediante compresión de imágenes. Sin embargo, ésta afecta sólo a la transmisión de datos y las tasas de buffering entre la computadora y la impresora; en algún punto, en el lado de la computadora o dentro del controlador del motor de impresión, se debe manejar la tasa de datos completa implicada por el número de pixeles a transferir a papel. Note que una imagen a color de un solo megapixel necesita casi 3 MB de almacenamiento. Muchas computadoras modernas tienen una memoria de video dedicada de este tamaño o mayor. Esto último permite que se almacene una imagen de pantalla completa y la transferencia de datos del CPU se limite a sólo los elementos que cambian de una imagen a la siguiente. La mayoría de los dispositivos I/O mencionados en la tabla 3.3 son de uso común en la actualidad; unos cuantos se consideran

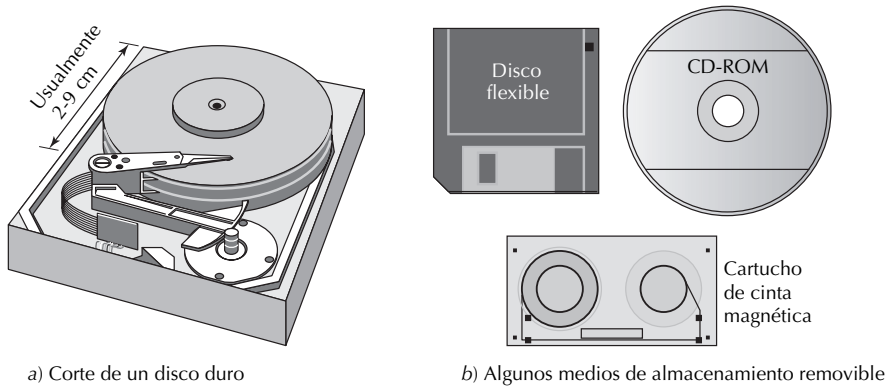


Figura 3.12 Unidades de memoria magnética y óptica.

“exóticos” o en etapa experimental (el guante para entrada 3D de ubicación/movimiento, esencias de entrada o salida, detección o generación de señales cerebrales). Sin embargo, incluso se espera que éstos logren estatuto principal en lo futuro.

Ciertos dispositivos de dos vías se pueden usar tanto para entrada como para salida. La jefatura entre éstos la tienen las unidades de almacenamiento masivo e interfases de red. Los discos magnéticos duros y flexibles, así como los discos ópticos (CD-ROM, CD-RW, DVD), funcionan con principios similares. Estos dispositivos (figura 3.12) leen o graban datos a lo largo de pistas concéntricas empaquetadas densamente en la superficie de un plato de disco rotatorio. En los discos duros, un mecanismo lectura/escritura, que se puede mover en forma radial en la parte adecuada de la pista pasa bajo la cabeza. Estos principios simples se han usado durante décadas. La característica que hace maravillas de la tecnología moderna a las unidades de almacenamiento masivo de la actualidad es su habilidad para detectar y manipular correctamente bits de datos empaquetados tan apretadamente juntos que una sola partícula de polvo bajo la cabeza puede borrar miles de bits. A lo largo de los años, el diámetro de las memorias de disco magnético se ha encogido más de diez veces, de decenas de centímetros a unos cuantos centímetros. Dado este encogimiento centuplicado en el área de grabación, los aumentos en capacidad desde meros megabytes a muchos gigabytes son todavía más notables. Las mejoras de rapidez son menos impresionantes y se lograron a través de actuadores más rápidos para mover las cabezas (que ahora necesitan recorrer distancias más cortas) y mayor rapidez de rotación. Los discos flexibles y otros de tipo removible son similares, excepto que, debido a la menor precisión (que conduce a densidad de grabación más baja) y rotación más lenta, son tanto más lentos como más pequeños en capacidad.

Cada vez más, las entradas llegan mediante líneas de comunicación, más que de dispositivos de entrada convencionales, y las salidas se escriben a archivos de datos a los que se accede a través de una red. No es raro que una impresora ubicada junto a una computadora en una oficina esté conectada a ella no directamente mediante un cable de impresora, sino por medio de una red de área local (LAN, por sus siglas en inglés). A través de una red de computadora, las máquinas se pueden comunicar entre ellas, así como con una diversidad de periféricos; por ejemplo, servidores de archivo, electrodomésticos, dispositivos de control y equipo de entretenimiento. La figura 3.13 muestra las dos características clave de banda ancha y latencia para una diversidad de sistemas de comunicación, desde los buses de ancho de banda alto que tienen menos de 1 m de largo, hasta redes de área ancha que abarcan el globo. Las computadoras y otros dispositivos se comunican a través de una red mediante unidades de interfaz de red. Se siguen protocolos especiales para garantizar que los diversos dispositivos de hardware puedan entender de manera correcta y consistente los datos que se transmiten. Módems de varios tipos (línea te-

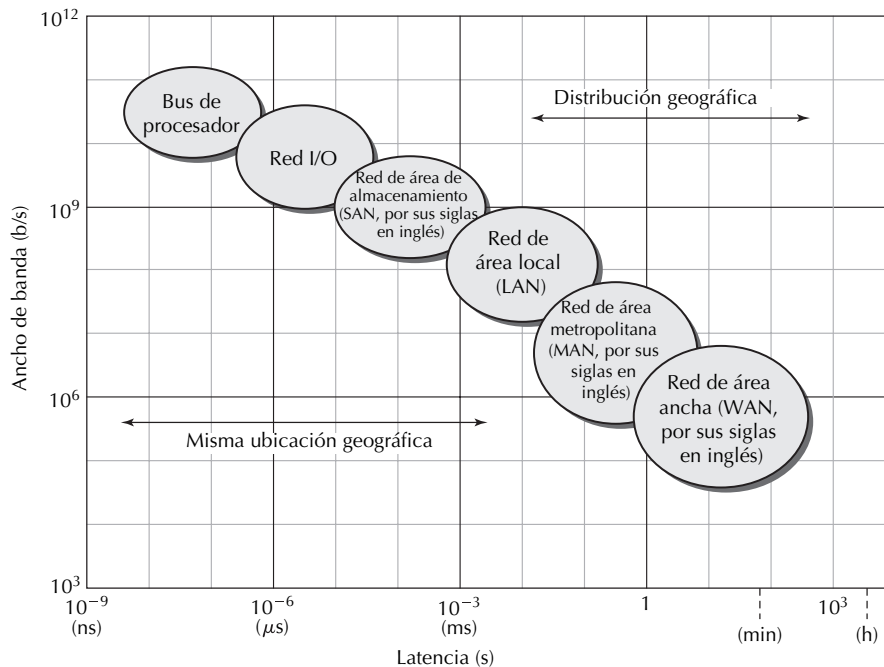


Figura 3.13 Latencia y ancho de banda característicos de diferentes clases de vínculos de comunicación.

telefónica, DSL, cable), tarjetas de interfaz de red, switches y “ruteadores” toman parte en la transmisión de datos desde una fuente hasta el destino deseado por medio de conexiones de muchos tipos diferentes (alambres de cobre, fibras ópticas, canales inalámbricos) y compuertas (*gateways*) de red.

3.6 Sistemas de software y aplicaciones

Las instrucciones que comprende el hardware de computadoras se codifican en cadenas de 0 y 1 y, por tanto, son indistinguibles de los números en los que pueden operar. Un conjunto de tales instrucciones constituye un *programa en lenguaje de máquina* que especifica un proceso computacional paso a paso (figura 3.14, extrema derecha). Las primeras computadoras digitales se programaban en lenguaje de máquina, un proceso tedioso que era aceptable sólo porque los programas de aquellos días eran bastante simples. Desarrollos subsecuentes condujeron a la invención del *lenguaje de ensamblado*, que permite la representación simbólica de programas en lenguaje de máquina, y lenguajes procedimentales de alto nivel que recuerdan la notación matemática. Estas representaciones más abstractas, con el software de traducción (*ensambladores* y *compiladores*) para conversión automática de programas a lenguaje de máquina, simplificaron el desarrollo de programas y aumentaron la productividad de los programadores. Gran parte de la computación a nivel usuario se realiza a través de notaciones de muy alto nivel, pues posee gran cantidad de poder expresivo para dominios específicos de interés. Los ejemplos incluyen procesamiento de textos, edición de imagen, dibujo de diagramas lógicos y producción de gráficas. Estos niveles de abstracción en la programación, y el proceso de ir de cada nivel al siguiente nivel inferior, se muestran en la figura 3.14.

Como se muestra en la figura 3.15, el software de computadora se puede dividir en clases de *software de aplicación* y *software de sistema*. El primero abarca a los procesadores de texto, programas

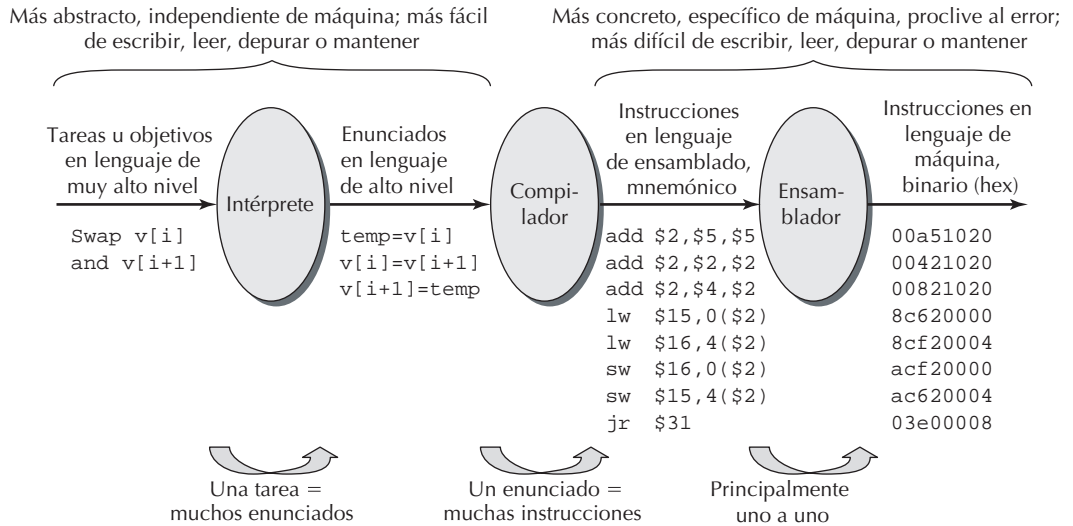


Figura 3.14 Modelos y abstracciones en programación.

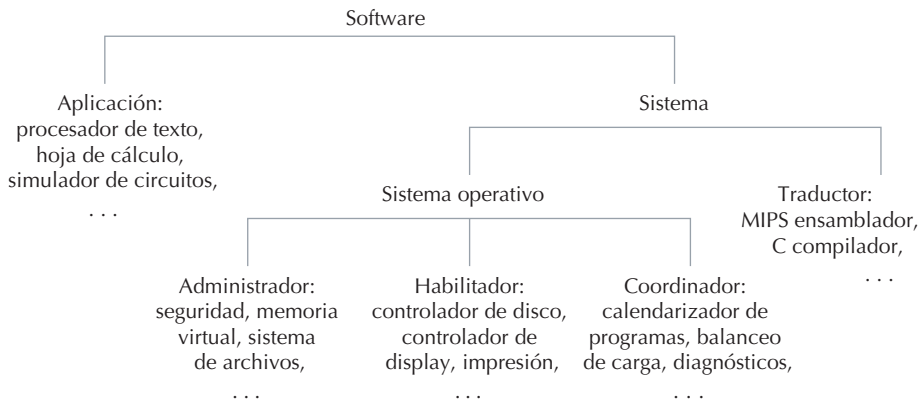


Figura 3.15 Categorización de software, con ejemplos en cada clase.

de hojas de cálculo, simuladores de circuito y muchos otros programas que se diseñan para enfrentar tareas específicas de interés para los usuarios. El segundo se divide en programas que traducen o interpretan instrucciones escritas en varios sistemas notacionales (como el lenguaje ensamblador MIPS o el lenguaje de programación C) y aquellos que ofrecen funciones administrativas, habilitadoras o coordinadoras para programas y recursos del sistema.

Existen funcionalidades que requieren la gran mayoría de los usuarios de computadora y, por tanto, se incorporan en el *sistema operativo*. Detalles de funciones de sistema operativo se encuentran en muchos libros de texto sobre la materia (vea las referencias al final del capítulo). En capítulos subsiguientes de este libro se discutirán brevemente algunos de estos temas: memoria virtual y algunos aspectos de seguridad en el capítulo 20, sistemas de archivos y controladores de disco en el capítulo 19, controladores de dispositivos I/O en los capítulos 21 y 22, y ciertos aspectos de coordinación en los capítulos 23 y 24.

PROBLEMAS

3.1 Definición de arquitectura de computadoras

Es el año 2010 y se le pide escribir un artículo, que ocupe una página, acerca de la arquitectura de computadoras para una enciclopedia infantil de ciencia y tecnología. ¿Cómo describiría la arquitectura de computadoras a los niños de primaria? Se le permite usar un diagrama, si fuera necesario.

3.2 La importancia de un arquitecto

Agregue una palabra más antes de cada una de las cuatro elipses de la figura 3.2.

3.3 El lugar central de la arquitectura de computadoras

En la figura 3.1, el diseñador de computadoras, o arquitecto, ocupa una posición central. Esto no es un accidente: nos gusta vernos o a nuestra profesión como algo fundamental. Testito es el mapa del mundo según lo dibujan los americanos (América en el centro, flanqueada por Europa, África y partes de Asia a la derecha, con el resto de Asia más Oceanía a la izquierda) y los europeos (América a la izquierda y todo Asia a la derecha).

- De haber sido éste un libro acerca de circuitos lógicos, habría visto la figura equivalente de la figura 3.1 con el diseñador lógico en medio. Describa lo que vería en las dos burbujas a cada lado.
- Repita la parte a) para un libro acerca de circuitos electrónicos.

3.4 Sistemas complejos hechos por la humanidad

Los sistemas de cómputo se encuentran entre los sistemas más complejos hechos por la humanidad.

- Mencione algunos sistemas que considere son más complejos que una moderna computadora digital. Elabore bajo sus criterios de complejidad.
- ¿Cuál de estos sistemas, para el caso de que existiera alguno, ha conducido a un campo de estudio separado en la ciencia o la ingeniería?

3.5 Múltiplos de unidades

Si supone que X es el símbolo de un múltiplo de potencia de 10 arbitrario en la tabla 3.1, ¿cuál es el máximo

error relativo si uno usa por equivocación X_b en lugar de X , o viceversa?

3.6 Computadoras embebidas

Una función que se necesita frecuentemente en las aplicaciones de control embebidas es la conversión analógica a digital (A/D). Estudie este problema y prepare un informe de dos páginas acerca de sus hallazgos. Este último debe incluir:

- Al menos una forma de realizar la conversión, incluido un diagrama de hardware.
- Descripción de una aplicación para la que se requiera la conversión A/D.
- Una discusión de la precisión del proceso de conversión y sus implicaciones.

3.7 Computadoras personales

Mencione todas las razones que usted crea que una computadora laptop o notebook es mucho más pequeña que una computadora de escritorio de potencia computacional comparable.

3.8 Supercomputadoras

Encuentre tanta información como pueda acerca de la supercomputadora más poderosa de que tenga información y escriba un informe de dos páginas acerca de ella. Comience su estudio con [Top500] e incluya lo siguiente.

- Criterios que conducen a clasificar la supercomputadora como la más poderosa.
- Compañía u organización que construyó la máquina y su motivación/consumidor.
- Identidad y potencia computacional relativa de su competidor más cercano.

3.9 Partes de una computadora

Mencione uno o más órganos humanos que tengan funciones similares a las asociadas con cada parte de la figura 3.7.

3.10 Historia de la computación digital

Charles Babbage, quien vivió hace dos siglos, es considerado el “abuelo” de la computación digital. Sin embargo, muchos creen que la computación digital tiene orígenes mucho más antiguos.

- Estudie el artículo [deSo84] y prepare un ensayo de dos páginas acerca del tema.
- Use una moderna calculadora electrónica y unas cuantas de las máquinas descritas en [deSo84], grafique las tendencias en tamaño, costo y rapidez de las máquinas calculadoras a lo largo de los siglos.

3.11 El futuro de las calculadoras de bolsillo/escritorio

El ábaco, todavía en uso en ciertas áreas remotas, resulta anacrónico en gran parte del mundo. La regla de cálculo se volvió obsoleta en un tiempo mucho más corto. ¿En qué crees que se convertirán las actuales calculadoras de bolsillo y escritorio? ¿Crees que todavía se usarán en el año 2020? Discute.

3.12 Máquina planeada de Babbage

Charles Babbage planeó construir una máquina que multiplicara números de 50 dígitos en menos de un minuto.

- Compare la rapidez de la máquina de Babbage con la de un calculador humano. ¿Puedes comparar los dos en cuanto a su confiabilidad?
- Repita las comparaciones de la parte *a*) con una moderna computadora digital.

3.13 Tendencias de costo

Al graficar el costo de una computadora por unidad de potencia computacional, se obtiene una curva que declina de forma aguda. Las computadoras de costo cero ya están aquí. ¿Puedes vislumbrar máquinas de costo negativo en lo futuro? Discute.

3.14 Variación de producción con tamaño de dado

La figura 3.9 y el ejemplo 3.1 muestran el efecto de aumentar el tamaño de dado de $1 \times 1 \text{ cm}^2$ a $2 \times 2 \text{ cm}^2$.

- Con las mismas suposiciones que las del ejemplo 3.1, calcule la producción y costo de dado relativo para dados cuadrados de 3×3 .
- Repita la parte *a*) para dados rectangulares de 2×4 .

3.15 Efectos de producción en el costo de dados

Una oblea que contiene 100 copias de un dado procesador complejo tiene un costo de producción de 900 dólares.

El área que ocupa cada procesador es de 2 cm^2 y la densidad de defecto es de $2/\text{cm}^2$. ¿Cuál es el costo de fabricación por dado?

3.16 Número de dados en una oblea

Considere una oblea circular de diámetro d . El número de dados cuadrados de lado u en la oblea está acotado por $\pi d^2/(4u^2)$. El número real será más pequeño debido a que existen dados incompletos en el borde.

- Argumente que $\pi d^2/(4u^2) - \pi d/(1.414u)$ representa una estimación bastante precisa del número de dados.
- Aplice la fórmula de la parte *a*) a las obleas que se muestran en la figura 3.9 para obtener una estimación del número de dados y determine el error en cada caso. Los dados son de 1×1 , 2×2 y $d = 14$.
- Sugiera y justifique una fórmula que funcionaría para dados no cuadrados $u \times v$ (por ejemplo, $1 \times 2 \text{ cm}^2$).

3.17 Tecnologías de procesador y memoria

Encuentre datos de rendimiento y capacidad en los más actuales chips procesador y memoria DRAM. Marque los puntos correspondientes en la figura 3.10. ¿Cuán bien encajan las líneas extrapoladas? ¿Estos nuevos puntos indican algún retardo en la tasa de progreso? Discuta.

3.18 Empaquetado de computadoras

Se planea construir una computadora paralela de 4 096 nodos. Éstos se organizan como una malla 3D de $16 \times 16 \times 16$, con cada nodo conectado a seis vecinos (dos a lo largo de cada dimensión). Se pueden encajar ocho nodos en un chip VLSI a la medida, y se pueden colocar 16 chips en una tarjeta de circuito impreso.

- Diseñe un esquema de partición para la computadora paralela que minimizará el número de vínculos fuera de chip, de tarjeta y de chasis.
- Al considerar el esquema de empaquetamiento de la figura 3.11a y la partición sugerida en la parte *a*), ¿sería capaz de acomodar canales de ocho bits de ancho?
- ¿El esquema de empaquetamiento 3D de la figura 3.11b ofrece algún beneficio para este diseño?

REFERENCIAS Y LECTURAS SUGERIDAS

- [Alla02] Allan, A. *et al.*, “2001 Technology Roadmap for Semiconductors”, *IEEE Computer*, vol. 35, núm. 1, pp. 42-53, enero 2002.
- [deSo84] de Solla Price, D., “A History of Calculating Machines”, *IEEE Micro*, vol. 4, núm. 1, pp. 22-52, febrero de 1984.
- [Lee02] Lee, T. H., “Vertical Leap for Microchips”, *Scientific American*, vol. 286, núm. 1, pp. 52-59, enero de 2002.
- [Raba96] Rabaey, J. M., *Digital Integrated Circuits: A Design Perspective*, Prentice Hall, 1996.
- [Rand82] Randell, B. (ed.), *The Origins of Digital Computers: Selected Papers*, Springer-Verlag, 3a. ed., 1982.
- [SIA00] Semiconductor Industry Association, *International Technology Roadmap for Semiconductors*, San Jose, CA, actualización 2000, <http://public.itrs.net/>
- [Stal03] Stallings, W., *Computer Organization and Architecture*, Prentice Hall, 6a. ed., 2003.
- [Tane01] Tanenbaum, A., *Modern Operating Systems*, Prentice Hall, 2a. ed., 2001.
- [Top500] Sitio de las 500 mejores supercomputadoras, sitio Web mantenido por la Universidad de Mannheim y la Universidad de Tennessee. <http://www.top500.org/>
- [Wilk95] Wilkes, M. V., *Computing Perspectives*, Morgan Kaufmann, 1995.
- [Wu00] Wu, L. *et al.*, “The Advent of 3D Package Age”, *Proceedings of the International Electronics Manufacturing Technology Symposium*, 2000, pp. 102-107.

■ CAPÍTULO 4

RENDIMIENTO DE COMPUTADORAS

“Rendimiento pico: nivel de rendimiento que se garantiza, es mayor que el rendimiento realmente logrado.”

Del folclor computacional, fuente desconocida

“El rápido dirige al lento, aun cuando el rápido esté equivocado.”

William Kahan

TEMAS DEL CAPÍTULO

- 4.1 Costo, rendimiento y costo/rendimiento
- 4.2 Definición de rendimiento de computadora
- 4.3 Mejora del rendimiento y ley de Amdahl
- 4.4 Medición del rendimiento contra modelado
- 4.5 Informe del rendimiento de computadoras
- 4.6 Búsqueda de mayor rendimiento

En los capítulos 1 al 3 se adquirieron los antecedentes necesarios para estudiar la arquitectura de computadoras. El último aspecto a considerar antes de entrar en detalles del tema es el rendimiento de las computadoras. Se tiende a igualar “rendimiento” con “rapidez”, pero esto último, en el mejor de los casos, resulta una visión simplista. Existen muchos aspectos del rendimiento. Comprender todos éstos le ayudarán a darse una idea de las diversas decisiones de diseño que se encuentran en los capítulos siguientes. Durante muchos años, el rendimiento ha sido la fuerza impulsora clave detrás de los avances en la arquitectura de computadoras. Todavía es muy importante, pero como consecuencia de que los procesadores modernos tienen rendimientos para prescindir de la mayoría de las aplicaciones que “corren la milla”, otros parámetros como costo, compactación y economía eléctrica están ganando rápidamente en significancia.

■ 4.1 Costo, rendimiento y costo/rendimiento

Una gran cantidad de trabajo en arquitectura de computadoras trata con los métodos para mejorar el rendimiento de máquina. En los siguientes capítulos encontrará ejemplos de tales métodos. En este sentido, todas las decisiones de diseño al construir computadoras, desde el diseño de conjunto de instrucciones hasta el uso de técnicas de implementación como *encauzamiento*, predicción de bifur-

cación, memorias caché y paralelismo, si no principalmente están motivadas por el mejoramiento del rendimiento, al menos se hacen con ese propósito. De acuerdo con lo anterior, es importante tener una definición operativa precisa para el concepto rendimiento, conocer sus relaciones con otros aspectos de la calidad y utilidad de las computadoras, y aprender cómo se puede cuantificar con propósitos de comparación y decisiones de comercialización.

Un segundo atributo clave de un sistema de cómputo es su costo. En cualquier año específico, probablemente se pueda diseñar y construir una computadora que sea más rápida que la más rápida de las computadoras actuales disponibles en el mercado. Sin embargo, el costo podría ser tan inalcanzable que quizá esta última máquina nunca se construya o se fabrique en cantidades muy pequeñas por agencias que estén interesadas en avanzar el estado del arte y no les importe gastar una cantidad exorbitante para lograr esa meta. Por ende, la máquina de mayor rendimiento que sea tecnológicamente factible puede nunca materializarse porque es *inefcaz en costo* (tiene una *razón costo/rendimiento* inaceptable debido a su alto costo). Sería simplista igualar el costo de una computadora con su precio de compra. En vez de ello, se debería intentar evaluar su *costo de ciclo de vida*, que incluye actualizaciones, mantenimiento, uso y otros costos recurrentes. Observe que una computadora que se compra por dos mil dólares tiene diferentes costos. Puede haberle costado 1 500 dólares al fabricante (por componentes de hardware, licencias de software, mano de obra, embarque, publicidad), y los 500 dólares restantes cubran comisiones de ventas y rentabilidad. En este contexto, podría llegar a costar cuatro mil dólares durante su tiempo de vida una vez agregado servicio, seguro, software adicional, actualizaciones de hardware, etcétera.

Con el propósito de apreciar que el rendimiento de la computadora es multifacético y que cualquier indicador aislado proporciona cuando mucho un cuadro aproximado, se usa una analogía con los aviones de pasajeros. En la tabla 4.1, seis aeronaves comerciales se caracterizan por su capacidad de pasajeros, rango de crucero, rapidez de crucero y precio de compra. Con base en los datos de la tabla 4.1, ¿cuál de esas aeronaves tiene el mayor rendimiento? Estaría justificado responder a esta pregunta con otra: ¿rendimiento desde el punto de vista de quién?

Un pasajero interesado en reducir su tiempo de viaje puede igualar el rendimiento con la rapidez de crucero. El *Concorde* claramente gana en este sentido (ignore el hecho de que esa aeronave ya no está en servicio). Observe que, debido al tiempo que a la aeronave le toma prepararse, despegar y aterrizar, la ventaja del tiempo de viaje es menor que la razón de rapidez. Ahora suponga que la ciudad de destino de un pasajero está a 8 750 km. Al ignorar los retardos pre y posvuelo por simplicidad, se encuentra que el DC-8-50 llegaría ahí en diez horas. El tiempo de vuelo del *Concorde* sería sólo de cuatro horas, pero algunas de sus ventajas desaparecen cuando se factoriza la parada obligatoria para reabastecer combustible. Por la misma razón, el DC-8-50 probablemente es mejor que el más rápido Boeing 747 o 777 para vuelos cuyas distancias superan el rango de este último.

■ **TABLA 4.1** Características clave de seis aeronaves de pasajeros: todas las cifras son aproximadas; algunas se relacionan con un modelo/configuración específico de la aeronave o son promedios del rango de valores citado.

Aeronave	Pasajeros	Rango (km)	Rapidez (km/h)	Precio*(\$M)
Airbus A310	250	8 300	895	120
Boeing 747	470	6 700	980	200
Boeing 767	250	12 300	885	120
Boeing 777	375	7 450	980	180
Concorde	130	6 400	2 200	350
DC-8-50	145	14 000	875	80

* Los precios se derivan mediante extrapolación y cierta adivinación. Con frecuencia, los aviones de pasajeros se venden con grandes descuentos sobre los precios de lista. Algunos modelos, como el ahora retirado *Concorde*, ya no se producen o nunca se vendieron en el mercado abierto.

Y esto fue sólo el enfoque del pasajero. Una línea aérea puede estar más interesada en el *rendimiento total*, que se define como el producto de la capacidad de pasajeros y rapidez (dentro de poco se tratará con estos conflictos de costo). Si las tarifas aéreas fuesen proporcionales a las distancias voladas, que en el mundo real no es así, el rendimiento total representaría el ingreso por venta de boletos de la aerolínea. Los seis aviones tienen rendimientos totales de 0.224, 0.461, 0.221, 0.368, 0.286 y 0.127 millones de pasajero-kilómetros por hora, respectivamente, siendo el Boeing 747 el que exhibe el mayor rendimiento total. Finalmente, el rendimiento desde el punto de vista de la Federal Aviation Administration se relaciona principalmente con el registro de seguridad de un avión.

Desde luego, el rendimiento nunca se ve aislado. Muy pocas personas consideraron que la ventaja de tiempo de viaje del *Concorde* valía en mucho su tarifa aérea. De igual modo, muy pocas aerolíneas estaban deseosas de pagar el mayor precio de compra del *Concorde*. Por esta razón, los indicadores combinados de rendimiento y costo son de interés. Suponga que el rendimiento se especifica mediante un indicador numérico, de modo que los valores más grandes de este indicador son preferibles (la rapidez de la aeronave es un ejemplo). El *costo/rendimiento*, que se define como el costo de una unidad de rendimiento, o su inverso, así como el rendimiento logrado por unidad de costo, se pueden usar para comparar *costo-beneficio* de varios sistemas. De este modo, igualar el rendimiento con el rendimiento total y el costo con el precio de compra, el coeficiente de mérito costo/rendimiento para las seis aeronaves en la tabla 4.1 son 536, 434, 543, 489, 1224 y 630, respectivamente, con valores más pequeños se reputan mejor.

Desde luego, la comparación anterior es bastante simplista; el costo de una aeronave para una aerolínea involucra no sólo su precio de compra sino también su economía de combustible, disponibilidad/precio de partes, frecuencia/facilidad de mantenimiento, costos relacionados con la seguridad (por ejemplo, seguros), etc. Para todo propósito práctico, el factor costo para un pasajero es simplemente la tarifa aérea. Observe que tal medición compuesta es incluso menos precisa que el rendimiento o el costo sólo porque incorpora dos factores difíciles de cuantificar.

Enseguida se hará una observación final en cuanto a la analogía de los aviones. El mayor rendimiento de un sistema es de interés sólo si uno puede beneficiarse verdaderamente de él. Por ejemplo, si usted viaja de Boston a Londres y el *Concorde* sólo despegue desde Nueva York, entonces su mayor rapidez puede haber no tenido significación para usted en relación con este viaje particular. En términos computacionales, esto es semejante a una nueva arquitectura de 64 bits que no ofrece beneficios cuando el volumen de sus aplicaciones se ha diseñado para máquinas más antiguas de 32 bits. De manera similar, si usted quiere ir de Boston a Nueva York, el tiempo de vuelo es una fracción tan pequeña del tiempo total empleado que no haría mucha diferencia si su avión es un Boeing 777 o un DC-8-50. Para el análogo computacional de esta situación, considere que a veces sustituir un procesador de computadora con una más rápida no tendrá un impacto significativo en el rendimiento porque éste se encuentra limitado por la memoria o el ancho de banda I/O.

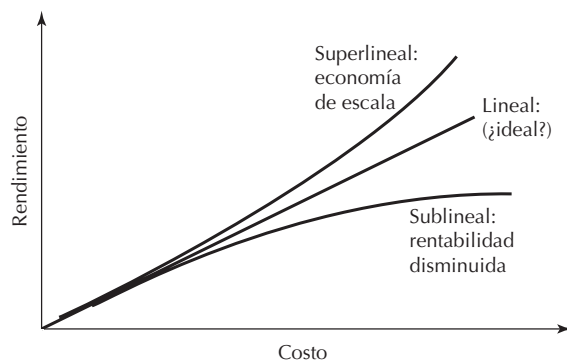


Figura 4.1 Mejora del rendimiento como función del costo.

Graficar el rendimiento contra el costo (figura 4.1) revela tres tipos de tendencia. El crecimiento superlineal del rendimiento con el costo indica *economía de escala*: si usted paga el doble de algo, obtiene más del doble de rendimiento. Éste fue el caso en los primeros días de las computadoras digitales cuando las supercomputadoras podían rendir significativamente más cálculos por dólar gastado que las máquinas más pequeñas. Sin embargo, en la actualidad, con frecuencia se observa una tendencia sublineal. Conforme se agrega más hardware a un sistema, se pierde parte del rendimiento teóricamente posible en mecanismos de gestión y coordinación necesarios para correr el sistema más complejo. Igualmente, un procesador de avanzada que cueste el doble de un modelo más antiguo puede no ofrecer el doble de rendimiento. Con este tipo de tendencia, pronto se alcanza un punto de *rentabilidad reducida* más allá de la cual la ulterior inversión no compra mucho en rendimiento. En este contexto, la mejora lineal en el rendimiento con costo se puede considerar un ideal por el cual luchar.

■ 4.2 Definición de rendimiento de computadora

Como usuarios, uno espera que una computadora de mayor rendimiento corra más rápido los programas de aplicaciones. De hecho, el tiempo de ejecución de los programas, ya sean cálculos de largo procesamiento o comandos simples a los que la computadora debe reaccionar de inmediato, es un indicador de rendimiento universalmente aceptado. En virtud de que un tiempo de ejecución más largo implica rendimiento más bajo, se puede escribir:

$$\text{Rendimiento} = 1/\text{tiempo de ejecución}$$

De este modo, el hecho de que una computadora ejecute un programa en la mitad del tiempo que le toma a otra máquina significa que tiene el doble de rendimiento. Todos los otros indicadores representan aproximaciones al rendimiento que se usan porque uno no puede medir o predecir los tiempos de ejecución para programas reales. Las posibles razones incluyen falta de conocimiento acerca de exactamente cuáles programas correrán en la máquina, el costo de transferir programas a un sistema nuevo por el solo propósito de evaluación comparativa, y la necesidad de valorar una computadora que todavía no se ha construido o de algún otro modo no está disponible para experimentación.

Como con la analogía de los aviones de la sección 4.1, existen otras visiones de rendimiento. Por ejemplo, un centro de cómputo que vende tiempo de máquina a una diversidad de usuarios puede considerar el *rendimiento total computacional*, la cantidad total de tareas realizadas por unidad de tiempo, como lo más relevante, porque ello afecta directamente los ingresos del centro. De hecho, la ejecución de ciertos programas puede ser retrasada de intención si ello conduce a un mejor rendimiento total global. La estrategia de un autobús cuyo conductor elige el orden de bajada de los pasajeros para minimizar el tiempo de servicio total es una buena analogía para esta situación. En este orden de ideas, el tiempo de ejecución y el rendimiento total no son completamente independientes en cuanto a que mejorar uno usualmente (mas no siempre) conduce a una mejora en el otro. Por esta razón, en el resto del texto, el enfoque se hará sobre la percepción de rendimiento del usuario, que es el inverso del tiempo de ejecución.

De hecho, un usuario puede estar preocupado no con el tiempo de ejecución del programa *per se*, sino con el *tiempo de respuesta* o el *tiempo de retorno* total, que incluye latencia adicional atribuible a decisiones de calendarización, interrupciones de trabajo, retardos por cola I/O, etc. A veces a esto se le refiere como *tiempo de reloj de pared*, porque se puede medir al echar un vistazo al reloj de pared al comenzar y al terminar una tarea. Para filtrar los efectos de tales factores enormemente variables y difíciles de cuantificar, en ocasiones se usa el *tiempo de ejecución de CPU* para definir el rendimiento percibido por el usuario:

$$\text{Rendimiento} = 1/\text{tiempo de ejecución de CPU}$$

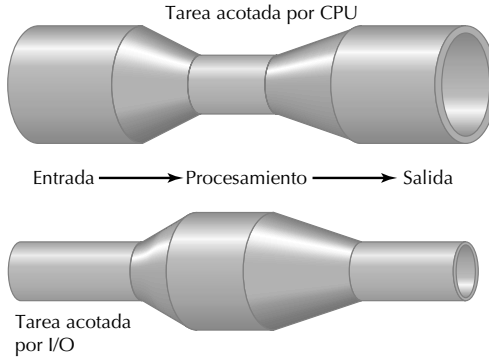


Figura 4.2 Analogía de tubería encauzamiento que muestra que el desequilibrio entre potencia de procesamiento y capacidades I/O conduce a un problema del rendimiento.

Este concepto vuelve la evaluación mucho más manejable en casos que conllevan métodos de evaluación analíticos, más que experimentales (vea la sección 4.4). Tal visión no conduce a imprecisión alguna para tareas de cálculo intenso que no involucren mucha I/O. Para tales *tareas acotadas por CPU*, la potencia de procesamiento representa el problema. Asimismo, las tareas acotadas por I/O serán mal atendidas si sólo se toma en cuenta el tiempo de ejecución de CPU (figura 4.2). Para un sistema balanceado sin problemas, no se erraría demasiado si se considera sólo el tiempo de ejecución de CPU en la evaluación del rendimiento. Observe que el equilibrio del sistema es esencial. Si uno sustituye el procesador de una máquina con un modelo que ofrece el doble de rendimiento, ello no duplicará el rendimiento global del sistema a menos que se realicen mejoras correspondientes en otras partes del sistema (memoria, bus del sistema, I/O, etc.). De paso, se menciona que duplicar el rendimiento del procesador no significa sustituir un procesador de x GHz con uno de $2x$ GHz; dentro de poco se verá que la frecuencia de reloj sólo es uno de los muchos factores que afectan el rendimiento.

Cuando uno compara dos máquinas M_1 y M_2 , la noción de rendimiento relativo entra en juego.

$$\begin{aligned} &(\text{rendimiento de } M_1)/(\text{rendimiento de } M_2) \\ &= \text{aceleración de } M_1 \text{ sobre } M_2 \\ &= (\text{tiempo de ejecución de } M_2)/(\text{tiempo de ejecución de } M_1) \end{aligned}$$

Observe que el rendimiento y el tiempo de ejecución varían en direcciones opuestas. Para mejorar el tiempo de ejecución, se debe reducirlo; mejorar el rendimiento significa elevarlo. Se tiende a favorecer el uso de “mejorar” sobre otros términos porque él permite aplicar un término común para muchas estrategias diferentes que impactan los indicadores de rendimiento, sin importar cuál indicador debe subir o bajar para un mejor rendimiento.

La medición de rendimiento comparativo apenas definida es una razón adimensional como 1.5 o 0.8. Indica que la máquina M_1 ofrece x veces el rendimiento, o es x veces más rápido que la máquina M_2 . Cuando $x > 1$, el rendimiento relativo se puede expresar en una de dos formas equivalentes:

$$\begin{aligned} &M_1 \text{ es } x \text{ veces más rápida que } M_2 \text{ (por ejemplo, 1.5 veces más rápida)} \\ &M_1 \text{ es } 100(x - 1)\% \text{ más rápida que } M_2 \text{ (por ejemplo, 50\% más rápida).} \end{aligned}$$

Fracasar al diferenciar estos dos métodos de presentación constituye un error bastante común. Así que, recuerde, una máquina que es 200% más rápida no es el doble de rápida sino tres veces más rápida. De manera más general: $y\%$ más rápido significa $1 + y/100$ veces más rápida.

Cada vez que se corre un programa específico, se ejecuta un número de instrucciones de máquina. Con frecuencia, este número es diferente de una corrida a otra, pero suponga que se conoce el número promedio de instrucciones que se ejecutan durante muchas corridas del programa. Observe que ese número puede tener poca relación con el número de instrucciones en el código del programa. Lo último

es el contador *instrucción estática*, mientras que aquí se está interesado en el contador *de instrucción dinámica*, que usualmente es mucho mayor que el contador estático debido a los ciclos y llamadas repetidas a ciertos procedimientos. La ejecución de cada instrucción toma cierto número de ciclos de reloj. De nuevo, el número es diferente para varias instrucciones y de hecho puede depender no sólo de la instrucción sino también del contexto (las instrucciones que se ejecutan antes y después de una instrucción específica). Suponga que también tiene un valor promedio para este parámetro. Finalmente, cada ciclo de reloj representa una duración fija de tiempo. Por ejemplo, el tiempo del ciclo de un reloj de 2 GHz es de 0.5 ns. El producto de estos tres factores produce una estimación del tiempo de ejecución de CPU para el programa:

$$\begin{aligned}\text{tiempo de ejecución de CPU} &= \text{instrucciones} \times (\text{ciclos por instrucción}) \times (\text{segundos por ciclo}) \\ &= \text{instrucciones} \times \text{CPI} / (\text{tasa de reloj})\end{aligned}$$

donde CPI significa “ciclos por instrucción” y la tasa de reloj, expresada en ciclos por segundo, es el inverso de “segundos por ciclo”.

El contador de instrucción de los tres parámetros, CPI y tasa de reloj no son completamente independientes, de modo que mejorar uno por un factor específico puede no conducir a una mejora global en el tiempo de ejecución por el mismo factor.

La cuenta de instrucción depende de la arquitectura del conjunto de instrucciones (cuáles instrucciones están disponibles) y cuán efectivamente las usa el programador o compilador. Los conflictos del conjunto de instrucciones se analizan en la parte 2 del libro.

CPI depende de la arquitectura del conjunto de instrucciones y de la organización del hardware. La mayoría de los conflictos organizacionales que influyen directamente los CPI se introducen en la parte 4 del libro, aunque en la parte 3 se cubren conceptos que también son relevantes.

La tasa de reloj depende de la organización de hardware y de la tecnología de implementación. Algunos aspectos de tecnología que afectan la tasa de reloj se cubren en los capítulos 1 a 3. Otros conflictos vendrán en las partes 3 y 4.

Para dar sólo un ejemplo de estas interdependencias, considere el efecto sobre el rendimiento de elevar las tasas de reloj. Si la tasa de reloj de un procesador Intel Pentium se mejora por un factor de 2, el rendimiento quizá mejorará pero no necesariamente por el mismo factor. Como consecuencia de que diferentes modelos del procesador Pentium usan sustancialmente el mismo conjunto de instrucciones (sin que las extensiones ocasionales introduzcan un factor si se continúan corriendo programas preexistentes), el rendimiento se duplicaría sólo si los CPI permanecen iguales. Sin embargo, desafortunadamente, elevar las tasas de reloj con frecuencia se acompaña con un aumento en los CPI. Las razones de esto último se aprenderán en la parte 4. Aquí sólo se puntualiza que una técnica para acomodar mayores tasas de reloj es dividir el proceso de ejecución de instrucciones en un gran número de pasos dentro de un *encauzamiento* más profundo. El castigo en ciclos de reloj por pérdida o limpieza de tal *encauzamiento*, que sería necesario en casos de dependencias de datos y control o fallos de caché, es proporcional a su profundidad. Por tanto, CPI constituye una función creciente de la profundidad del *encauzamiento*.

El efecto de la tasa de reloj en el rendimiento es incluso más difícil de juzgar cuando uno compara máquinas con diferentes arquitecturas de conjunto de instrucciones. La figura 4.3 usa una analogía para traspassar esta dificultad. Duplicar la tasa de reloj es el análogo de una persona que da pasos el doble de rápido que otra. Sin embargo, si la persona que da pasos más rápidos necesita cinco veces más de éstos para ir desde el punto A hasta el punto B, su tiempo de viaje será 2.5 veces más largo. Con base en esta analogía, no es de sorprender que el vendedor de un procesador de x GHz afirme la ventaja de rendimiento sobre el procesador de $2x$ GHz de otra compañía. Que la afirmación sea cierta es otra historia; más tarde se verán ejemplos de cómo las afirmaciones del rendimiento pueden ser engañosas o totalmente falsas.

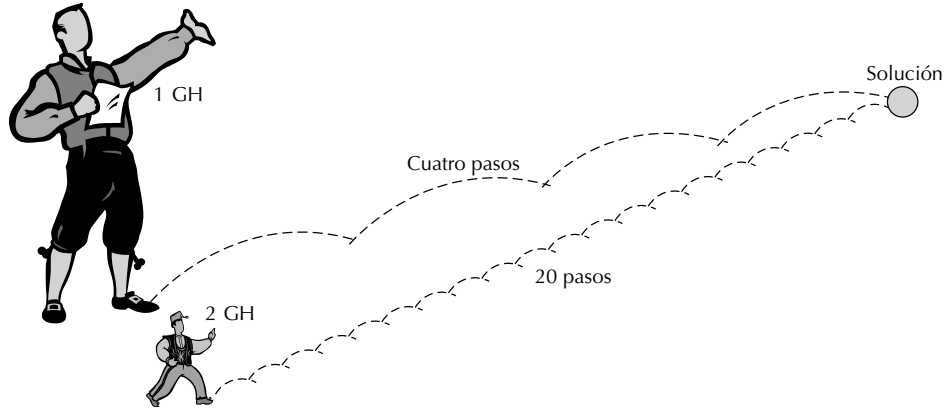


Figura 4.3 Los pasos más rápidos no necesariamente significan tiempo de viaje más corto.

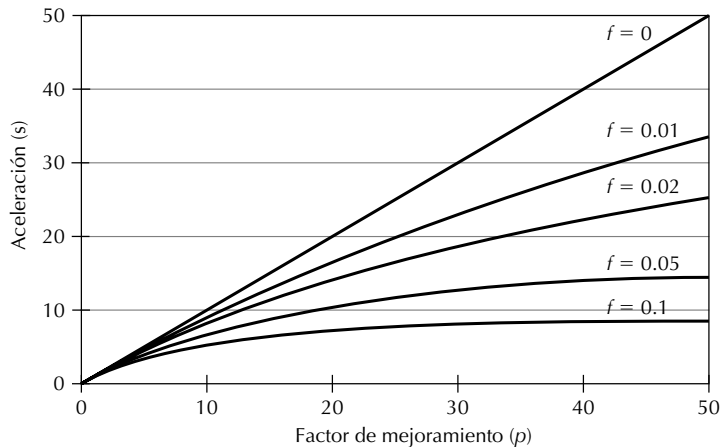


Figura 4.4 Ley de Amdahl: aceleración lograda si una fracción f de una tarea no se ve afectada y la parte restante $1 - f$ corre p veces más rápido.

4.3 Mejora del rendimiento y ley de Amdahl

Gene Amdahl, un arquitecto de las primeras computadoras IBM que después fundó la compañía que lleva su nombre, formuló su famosa ley (figura 4.4) para señalar algunas limitaciones del procesamiento paralelo. Él afirmó que los programas contenían ciertos cálculos que eran inherentemente secuenciales y, por tanto, no se podían acelerar mediante procesamiento paralelo. Si f representa la fracción del tiempo de corrido de un programa debido a tales cálculos no paralelizables, incluso suponiendo que el resto del programa disfrute la aceleración perfecta de p cuando corre en p procesadores, la aceleración global sería:

$$s = \frac{1}{f + (1 - f)/p} \leq \min\left(p, \frac{1}{f}\right) \quad [\text{fórmula de aceleración de Amdahl}]$$

Aquí, el número 1 en el numerador representa el tiempo de corrida original del programa y $f + (1 - f)/p$ constituye el tiempo de ejecución mejorado del programa con p procesadores. Lo último

es la suma del tiempo de ejecución para la fracción f no paralelizable y la fracción restante $1 - f$, que ahora corre p veces más rápido. Observe que la aceleración s no puede superar p (*aceleración lineal* lograda por $f = 0$) o $1/f$ (máxima aceleración para $p = \infty$). En consecuencia, para $f = 0.05$, uno nunca puede esperar lograr una aceleración mayor que 20, sin importar cuántos procesadores se usen.

A pesar de su formulación original en términos de la aceleración que es posible con p procesadores, la ley de Amdahl es mucho más general y se puede aplicar a cualquier situación que no involucre cambios en el tiempo de ejecución para una fracción f de un programa y mejora por un factor p (no necesariamente un entero) para la parte restante. Esta interpretación general sugiere que si se deja una parte de un programa que represente una fracción f de su tiempo de ejecución invariable, ninguna cantidad de mejora para la fracción restante $1 - f$ producirá una aceleración mayor que $1/f$. Por ejemplo, si la aritmética de punto flotante (*floating-point*) representa $1/3$ del tiempo de ejecución de un programa y se mejora sólo la unidad de punto flotante (es decir, $f = 2/3$), la aceleración global no puede superar 1.5, sin importar cuánto más rápido se vuelva la aritmética de punto flotante.

Ejemplo 4.1: Uso de la ley de Amdahl en diseño Un chip procesador se usa para aplicaciones en las que 30% del tiempo de ejecución se gasta en suma de punto flotante, 25% en multiplicación de punto flotante y 10% en división de punto flotante. Para el nuevo modelo de procesador, el equipo de diseño se ha topado con tres posibles mejoras, y cada una cuesta casi lo mismo en esfuerzo de diseño y fabricación. ¿Cuál de estas mejoras se debe elegir?

- Rediseñar el sumador de punto flotante para hacerlo el doble de rápido.
- Rediseñar el multiplicador de punto flotante para hacerlo tres veces más rápido.
- Rediseñar el divisor de punto flotante para hacerlo diez veces más rápido.

Solución: Se puede aplicar la ley de Amdahl a las tres opciones con $f = 0.7$, $f = 0.75$ y $f = 0.9$, respectivamente, para la fracción no modificada en los tres casos.

- Aceleración para rediseño de sumador $= 1/[0.7 + 0.3/2] = 1.18$
- Aceleración para rediseño de multiplicador $= 1/[0.75 + 0.25/3] = 1.20$
- Aceleración para rediseño de divisor $= 1/[0.9 + 0.1/10] = 1.10$

Por tanto, rediseñar el multiplicador de punto flotante ofrece la mayor ventaja de rendimiento, aunque la diferencia con el rediseño de sumador de punto flotante no es grande. A partir de este ejemplo se aprende una lección: la aceleración significativa del divisor no vale la pena el esfuerzo debido a la relativa rareza de las divisiones de punto flotante. De hecho, incluso si se pudiesen realizar divisiones infinitamente rápidas, la aceleración alcanzada todavía sería de sólo 1.11.

Ejemplo 4.2: Uso de la ley de Amdahl en administración Los miembros de un grupo universitario de investigación frecuentemente van a la biblioteca del campus para leer o copiar artículos publicados en revistas técnicas. Cada viaje a la biblioteca les toma 20 minutos. Con el propósito de reducir ese tiempo, un administrador ordena suscripciones para algunas revistas que representan 90% de los viajes a la biblioteca. Para estas revistas, que ahora se conservan en la biblioteca privada del grupo, el tiempo de acceso se redujo a dos minutos en promedio.

- ¿Cuál es la aceleración promedio para acceder a los artículos técnicos debido a las suscripciones?
- Si el grupo tiene 20 miembros y cada uno realiza en promedio dos viajes semanales a la biblioteca del campus, determine el gasto anual que es financieramente justificable para tomar las suscripciones. Suponga 50 semanas laborales al año y un costo promedio de 25 dólares/h para un tiempo de investigación.

Solución: Se puede aplicar la ley Amdahl a esta situación, donde 10% de los accesos permanecen invariables ($f = 0.1$) y el restante 90% se acelera por un factor de $p = 20/2 = 10$.

- a) Aceleración en el tiempo de acceso a los artículos = $1/[0.1 + 0.9/10] = 5.26$
- b) El tiempo que ahorran las suscripciones es $20 \times 2 \times 50 \times 0.9(20 - 2) = 32\,400 \text{ min} = 540 \text{ h}$ que representa una recuperación de costo de $540 \times 25 = 13\,500$ dólares; esta es la cantidad que se puede justificar financieramente por el costo de las suscripciones.

Nota: Este ejemplo es el análogo de usar una memoria caché rápida cerca del procesador con la meta de acceder más rápidamente a los datos que se usan con más frecuencia. Los detalles se cubrirán en el capítulo 18.

4.4 Medición del rendimiento contra modelado

El método más seguro y confiable de evaluar el rendimiento es correr programas reales de interés en máquinas candidatas y medir los tiempos de ejecución o de CPU. La figura 4.5 muestra un ejemplo con tres máquinas diferentes que se evalúan en seis programas. Con base en los datos de evaluación que se muestran, la máquina 3 claramente queda a la cabeza porque tiene el tiempo de ejecución más corto para los seis programas. Sin embargo, el resultado no es tan claro, como es el caso entre las máquinas 1 y 2 de la figura 4.5. La máquina 1 es más rápida que la máquina 2 para dos de los programas y más lenta para los otros cuatro. Si tuviese que elegir entre las máquinas 1 y 2 (por decir, porque la máquina 3 es mucho más cara o no satisface algún otro requisito importante), podría agregar pesos a los programas y elegir la máquina para que la *suma ponderada de tiempos de ejecución* sea más pequeña. El peso para un programa podría ser el número de veces que se ejecuta por mes (con base en datos recopilados o una predicción). Si, por ejemplo, los seis programas se ejecutan el mismo número de veces, y, por tanto, tienen pesos iguales, la máquina 2 tendría una ligera ventaja sobre la máquina 1. Si, por otra parte, los programas B o E constituyen la mayor parte de la *carga de trabajo*, la máquina 1 probablemente prevalecerá. En la sección 4.5 se elaborarán métodos para resumir o informar el rendimiento, y las trampas asociadas.

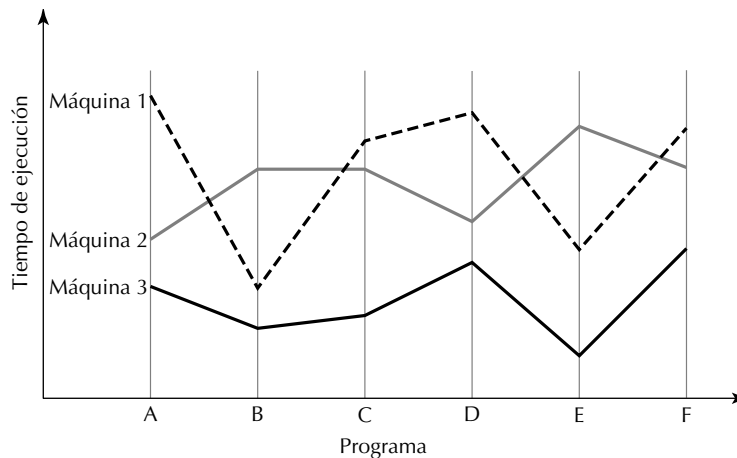


Figura 4.5 Tiempos de ejecución de seis programas en tres máquinas.

La experimentación con programas reales en máquinas reales no siempre es factible o económicamente viable. Si usted planea comprar hardware, las máquinas pretendidas pueden no estar a su alcance para una experimentación intensiva. De manera similar, es posible que los programas que correrá no estén disponibles o incluso no se conozcan. Es posible que tenga una idea de que correrá programas de cierto tipo (pago de nómina de la compañía, solucionador de ecuaciones lineales, diseño gráfico, etc.); algunos de los programas requeridos tendrán que diseñarse en casa y otros serán subcontratados. Recuerde que debe considerar no sólo las necesidades actuales sino también las futuras. En tales casos, la evaluación puede basarse en *programas de prueba (benchmarking)* o en *modelado analítico*.

Benchmarking

Los *benchmarks* son programas reales o sintéticos que se seleccionan o diseñan para evaluación comparativa del rendimiento de una máquina. Una suite *benchmark* representa una colección de tales programas que tiene la intención de representar toda clase de aplicaciones y hacer fracasar cualquier intento por diseñar hardware que tendría buen desempeño en un programa *benchmark* específico más limitado (a esto último se le conoce como *diseño de benchmarks*). Desde luego, los resultados del *benchmarking* sólo son relevantes para el usuario si los programas en la suite recuerdan los programas que el usuario correrá. Los *benchmarks* facilitan la comparación a través de diferentes plataformas y clases de computadoras. También hacen posible que los vendedores de computadoras y empresas independientes evalúen muchas máquinas antes de su entrada al mercado y publicar los resultados del *benchmarking* para beneficio de los usuarios. De esta forma, el usuario no necesitará realizar *benchmarking*.

Los *benchmarks* tienen la intención principal de usarse cuando el hardware a evaluar y los compiladores relevantes necesarios para correr los programas en la suite ya están disponibles. La mayoría de los compiladores tienen capacidades de optimización que se pueden encender o apagar. Usualmente, para evitar afinar el compilador de modo diferente para cada programa en la suite, se requiere que toda la suite de *benchmark* se corra con un solo conjunto de banderas de optimización. También es posible usar una suite *benchmark*, en especial una con programas más cortos, para la evaluación de máquinas o compiladores todavía no disponibles. Por ejemplo, los programas en la suite pueden ser compilados a mano y los resultados presentarse a un simulador de software del hardware a desarrollar. En este contexto, se pueden extraer cuentas de instrucción del código compilado a mano y usarse para evaluar el rendimiento en la forma indicada bajo el modelado analítico que sigue.

Una suite *benchmark* bastante popular para evaluar estaciones de trabajo y servidores incluye programas enteros (*integer*) y de punto flotante y está desarrollado por la Standard Performance Evaluation Corporation (SPEC, Corporación de Evaluación de Rendimiento Estándar). La versión 2000 de la suite *benchmark* SPEC CPU, conocida como SPECint2000 por la colección de 12 programas enteros y SPECfp2000 por otro conjunto de 12 programas de punto flotante, se caracteriza en la tabla 4.2. En lugar de proporcionar datos de tiempo de ejecución absolutos, es común informar cuánto más rápido

■ **TABLA 4.2** Resumen de características de la suite *benchmark* SPEC CPU2000.

Categoría	Tipos de programa	Ejemplos de programa	Líneas de código
SPECint2000	Programas C (11)	Compresión de datos, compilador de lenguaje C	0.7k a 193k
	Programa C++ (1)	Visualización de computadora (trazado de rayo)	34.2k
SPECfp2000	Programas C (4)	Gráficos 3D, química computacional	1.2k a 81.8k
	Programas Fortran77 (6)	Modelado de agua poco profunda, solucionador multirejilla	0.4k a 47.1k
	Programas Fortran90 (4)	Procesamiento de imagen, método de elemento finito	2.4k a 59.8k

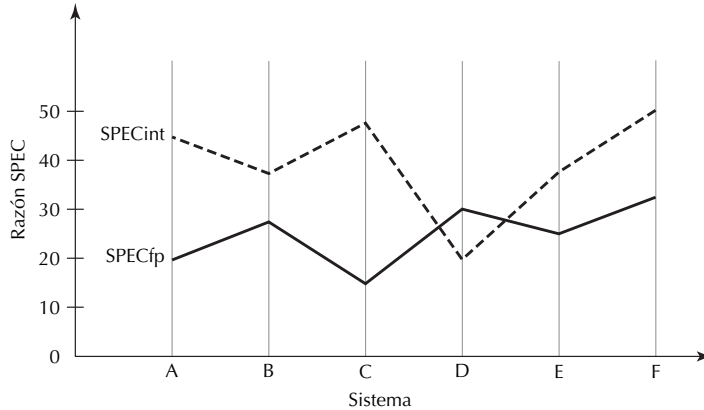


Figura 4.6 Ejemplo de esquematización gráfica de resultados de *benchmark* SPEC.

corrió un programa una máquina en comparación con alguna máquina base; mientras más grande sea esta *razón SPEC*, mayor es el rendimiento de la máquina. Luego se pueden graficar las razones calculadas por separado para SPECint y SPECfp para visualizar las diferencias entre muchas máquinas, o para la misma máquina con diferentes compiladores o tasas de reloj. La figura 4.6 muestra un ejemplo de resultados para seis diferentes sistemas, donde uno de éstos puede ser la combinación de un procesador con una tasa específica de reloj y tamaño caché/memoria, compiladores C y Fortran particulares con establecimiento de optimización específico, y capacidades I/O conocidas.

Ejemplo 4.3: Benchmarks de rendimiento Usted es ingeniero en Outtel, una nueva compañía que aspira a competir con Intel a través de su nueva tecnología de procesadores que funciona mejor que el último procesador Intel por un factor de 2.5 en instrucciones de punto flotante. Para lograr este nivel de rendimiento punto flotante, el equipo de diseño efectuó cierta negociación que condujo a 20% promedio de aumento en los tiempos de ejecución de todas las otras instrucciones. Usted está a cargo de elegir los *benchmarks* que mostrarían el límite de rendimiento de Outtel.

- ¿Cuál es la fracción mínima requerida f de tiempo gastado en instrucciones de punto flotante en un programa en el procesador Intel para mostrar una aceleración de 2 o mejor para Outtel?
- Si en el procesador Intel el tiempo de ejecución de una instrucción de punto flotante es en promedio tres veces tan largo como las otras instrucciones, ¿qué significa la fracción en su respuesta a la parte a en términos de la mezcla de instrucciones en el *benchmark*?
- ¿Qué tipo de *benchmark* elegiría Intel para contrarrestar las afirmaciones de su compañía?

Solución: Use una forma generalizada de la fórmula de Amdahl en la que una fracción f se acelere por un factor específico y el resto se detenga por otro factor.

- Factor de aceleración = 2.5, factor de frenado = 1.2 $\Rightarrow 1/[1.2(1 - f) + f/2.5] \geq 2 \Rightarrow f \geq 0.875$
- Sea la mezcla de instrucciones x punto flotante y $1 - x$ otra. Entonces, el tiempo de ejecución total es proporcional a $3x + (1 - x) = 2x + 1$. De modo que la fracción de tiempo de ejecución debido a operaciones de punto flotante es $3x/(2x + 1)$. Requerir que esta fracción sea mayor que o igual a 0.875 conduce a $x \geq 70\%$ (fracción punto flotante en mezcla de instrucciones).
- Intel intentaría mostrar un retardo para Outtel: $1/[1.2(1 - f) + f/2.5] < 1 \Rightarrow f < 0.125$. En términos de la mezcla de instrucciones, esto significa $3x/(2x + 1) < 0.125$ o $x < 4.5\%$.

Estimación de rendimiento

Es posible estimar el rendimiento de máquina sin hacer uso de observación directa del comportamiento de programas reales o *benchmarks* ejecutados en hardware real. Los métodos varían desde las estimaciones simplistas hasta el uso de modelos muy detallados que capturan los efectos de las características del diseño en hardware y software. Los modelos de rendimiento son de dos tipos: *Modelos analíticos* y *modelos de simulación*. Los primeros usan formulaciones matemáticas para relacionar el rendimiento con algunos parámetros claves, observables y cuantificables del sistema o la aplicación. Los segundos básicamente imitan el comportamiento del sistema, con frecuencia a un nivel de abstracción superior para conservar bajo observación el tamaño del modelo y su tiempo de procesamiento. Los resultados obtenidos por cualquier modelo sólo son tan buenos como la confiabilidad del modelo para representar capacidades, limitaciones e interacciones del mundo real. Es un error pensar que un modelo más detallado necesariamente ofrece una estimación de rendimiento más precisa. De hecho, la complejidad del modelo a veces oculta la comprensión y, por, ende conduce a una incapacidad para ver cómo el efecto de la imprecisión en la estimación de los parámetros del modelo puede afectar los resultados finales.

El modelo de estimación de rendimiento más simple es aquel que produce el *rendimiento pico* del sistema, se denomina así porque representa el nivel de rendimiento absoluto más elevado que uno puede esperar extraer del sistema. El rendimiento pico de una computadora es como la rapidez máxima de un automóvil y puede ser tan insignificante como un coeficiente de mérito para propósitos de comparación. Con frecuencia, el rendimiento pico se expresa en unidades de *instrucciones por segundo* o IPS, y se prefieren MIPS y GIPS para mantener los números pequeños. La ventaja del rendimiento pico es que es fácil de determinar y notificar. Para aplicaciones científicas y de ingeniería que involucran principalmente cálculos en punto flotante, se usan como unidad las *operaciones de punto flotante por segundo* (FLOPS, por sus siglas en inglés), de nuevo con megaflops (MFLOPS, por sus siglas en inglés) y gigaflops (GFLOPS, por sus siglas en inglés) como los favoritos. Una máquina logra su rendimiento pico para un programa construido artificialmente que incluye sólo instrucciones del tipo más rápido. Por ejemplo, en una máquina que tiene instrucciones que toman uno y dos ciclos de reloj para su ejecución, el rendimiento pico se logra si el programa usa exclusivamente instrucciones de un ciclo, acaso con unas cuantas instrucciones de dos ciclos lanzadas por ahí, si es necesario, para formar ciclos y otros programas constructores.

Un poco más detallado, y también más realista, es un análisis basado en CPI promedio (CPI se definió en la sección 4.2). Los CPI promedio se pueden calcular de acuerdo con una mezcla de instrucciones obtenida a partir de estudios experimentales. Tales estudios pueden examinar gran cantidad de programas usuales para determinar la proporción de varias clases de instrucciones, expresadas como fracciones que suman 1. Por ejemplo, la tabla 4.3 ofrece mezclas de instrucciones típicas. Si se eligen clases de instrucciones tales que todas las instrucciones en la misma clase tienen el mismo CPI, se pueden calcular los CPI promedio de las fracciones correspondientes:

$$\text{CPI promedio} = \sum_{\text{toda clase de instrucción}} (\text{Fracción clase } i) \times (\text{CPI clase } i)$$

■ **TABLA 4.3** Uso de frecuencia, en porcentaje, para diversas clases de instrucción en cuatro aplicaciones representativas.

Aplicación → Clase de instrucción ↓	Compresión de datos	Compilador lenguaje C	Simulación de reactor nuclear	Modelado de movimiento atómico
A: Load/Store	25	37	32	37
B: Integer arithmetic	32	28	17	5
C: Shift/Logical	16	13	2	1
D: Floating-point	0	0	34	42
E: Branch	19	13	9	10
F: Todas las demás	8	9	6	4

Una vez que se conocen los CPI promedio, se puede pretender que todas las instrucciones tengan estos CPI comunes y usar la fórmula

$$\text{Tiempo de ejecución CPU} = \text{instrucciones} \times (\text{CPI promedio})(\text{tasa de reloj})$$

derivada en la sección 4.2 para estimación de rendimiento.

Ejemplo 4.4: Cálculos de CPI e IPS Considere dos diferentes implementaciones hardware M_1 y M_2 del mismo conjunto de instrucciones. Existen tres clases de instrucciones en el conjunto de instrucciones: F, I y N. La tasa de reloj de M_1 es 600 MHz. El ciclo de reloj de M_2 es 2 ns. Los CPI promedio para las tres clases de instrucciones en M_1 y M_2 son los siguientes:

Clase	CPI para M_1	CPI para M_2	Comentarios
F	5.0	4.0	Floating-point
I	2.0	3.0	Integer arithmetic
N	2.4	2.0	No aritmético

- ¿Cuáles son los rendimientos pico de M_1 y M_2 en MIPS?
- Si 50% de todas las instrucciones ejecutadas en cierto programa son de la clase N y el resto se dividen por igual entre F e I, ¿cuál máquina es más rápida y por qué factor?
- Los diseñadores de M_1 planean rediseñar la máquina para mejor rendimiento. Con las suposiciones de la parte b), ¿cuál de las siguientes opciones de rediseño tiene el mayor impacto de rendimiento y por qué?
 - Usar una unidad de punto flotante más rápida con el doble de rapidez (CPI clase F = 2.5).
 - Añadir un segundo ALU entero para reducir los CPI enteros a 1.20.
 - Usar lógica más rápida que permita una tasa de reloj de 750 MHz con los mismos CPI.
- Los CPI dados incluyen el efecto de fallos de caché de instrucción a una tasa promedio de 5%. Cada uno de ellos impone un castigo de diez ciclos (es decir: suma 10 a los CPI efectivos de la instrucción, lo que provoca el fallo, o 0.5 ciclos por instrucción en el promedio). Una cuarta opción de rediseño es usar una caché de instrucción más grande que reduciría la tasa de fallos de 5% a 3%. ¿Cómo se compara esto con las tres opciones de la parte c)?
- Caracterice programas de aplicación que correrían más rápido en M_1 que en M_2 ; esto es: diga tanto como pueda acerca de la mezcla de instrucciones en tales aplicaciones. *Sugerencia:* Sean x , y y $1 - x - y$ la fracción de instrucciones que pertenecen a las clases F, I y N, respectivamente.

Solución

- MIPS pico para $M_1 = 600/2.0 = 300$ (suponga todos clase I)
MIPS pico para $M_2 = 500/2.0 = 250$ (suponga todos clase N)
- CPI promedio para $M_1 = 5.0/4 + 2.0/4 + 2.4/2 = 2.95$
CPI promedio para $M_2 = 4.0/4 + 3.8/4 + 2.0/2 = 2.95$
Los CPI promedio son iguales, de modo que M_1 es 1.2 veces más rápido que M_2 (razón de tasas de reloj).
1. CPI promedio = $2.5/4 + 2.0/4 + 2.4/2 = 2.325$; MIPS para opción 1 = $600/2.325 = 258$
2. CPI promedio = $5.0/4 + 1.2/4 + 2.4/2 = 2.75$; MIPS para opción 2 = $600/2.75 = 218$
3. MIPS para opción 3 = $750/2.95 = 254 \Rightarrow$ opción 1 tiene el mayor impacto.
- Con el caché más grande, todos los CPI se reducen por 0.2 debido a la tasa de fallo de caché más baja.
CPI promedio = $4.8/4 + 1.8/4 + 2.2/2 = 2.75 \Rightarrow$ opción 4 es comparable a la opción 2.

e) $CPI \text{ promedio para } M_1 = 5.0x + 2.0y + 2.4(1 - x - y) = 2.6x - 0.4y + 2.4$
 $CPI \text{ promedio para } M_2 = 4.0x + 3.8y + 2.0(1 - x - y) = 2x + 1.8y + 2$
Se buscan condiciones bajo las que $600/(2.6x - 0.4y + 2.4) > 500/(2x + 1.8y + 2)$. Por tanto, M_1 rinde mejor que M_2 para $x/y < 12.8$. Hablando burdamente, M_1 lo hace mejor a menos que exista aritmética de punto flotante excesiva, por lo que M_1 es poco más lento, o muy poca aritmética entera, para lo que M_2 es más lento (las instrucciones de clase N son inmateriales porque se ejecutan a la misma rapidez en ambas máquinas).

Ejemplo 4.5: Las calificaciones de MIPS pueden ser engañosas Considere dos compiladores que producen código de máquina para que un programa específico corra en la misma máquina. Las instrucciones de la máquina se dividen en clases A ($CPI = 1$) y clase B ($CPI = 2$). Los programas en lenguaje de máquina producidos por los dos compiladores conducen a la ejecución del siguiente número de instrucciones de cada clase:

Clase	Instrucciones para compilador 1	Instrucciones para compilador 2	Comentarios
A	600M	400M	$CPI = 1$
B	400M	400M	$CPI = 2$

- a) ¿Cuáles son los tiempos de ejecución de los dos programas, si se supone un reloj de 1 GHz?
- b) ¿Cuál compilador produce código más rápido y por qué factor?
- c) ¿Cuál salida de lenguaje de máquina de compilador corre a tasa MIPS mayor?

Solución

- a) Tiempo de ejecución para la salida del compilador 1 = $(600M \times 1 + 400M \times 2)/10^9 = 1.4 \text{ s}$
Tiempo de ejecución para la salida del compilador 2 = $(400M \times 1 + 400M \times 2)/10^9 = 1.2 \text{ s}$
- b) El código producido por el compilador 2 es $1.4/1.2 = 1.17$ veces más rápido que el compilador 1.
- c) $CPI \text{ promedio para la salida del compilador 1} = (600M \times 1 + 400M \times 2)/1000M = 1.4$
 $CPI \text{ promedio para la salida del compilador 2} = (400M \times 1 + 400M \times 2)/800M = 1.5$
De este modo, la calificación MIPS del compilador 1, que es $1000/1.4 = 714$, es poco mayor que la del compilador 2, que es $1000/1.5 = 667$, aun cuando, de acuerdo con los resultados de la parte a), la salida del compilador 1 es decididamente inferior.

■ **4.5 Informe del rendimiento de computadoras**

Incluso con el mejor método elegido para medir o modelar desempeño, se debe tener cuidado en la interpretación y notificación de los resultados. En esta sección se revisan algunas de las dificultades para separar los datos de rendimiento en un solo indicador numérico.

Considere los tiempos de ejecución para tres programas A, B y C en dos máquinas diferentes X y Y, que se muestran en la tabla 4.4. Los datos indican que, para el programa A, la máquina X es diez veces más rápida que la máquina B, mientras que tanto para B como para C, lo opuesto es cierto. El primer intento para resumir estos datos de rendimiento se halla al encontrar el promedio de las tres aceleraciones y valorar que la máquina Y es en promedio $(0.1 + 10 + 10)/3 = 6.7$ veces más rápida que la máquina X. Sin embargo, esto último es incorrecto. La última hilera de la tabla 4.4 muestra los tiempos de ejecución total para los tres programas y una aceleración global de 5.6, que sería la aceleración correcta para informar si estos programas corren el mismo número de veces dentro de la carga de trabajo

■ **TABLA 4.4** Tiempos de ejecución medidos o estimados para tres programas.

	Tiempo en máquina X	Tiempo en máquina Y	Aceleración de Y sobre X
Programa A	20	200	1.0
Programa B	1 000	100	10.0
Programa C	1 500	150	10.0
Los tres programas	2 520	450	5.6

normal. Si esta última condición no se mantiene, entonces la aceleración global se debe calcular con el uso de sumas ponderadas, más que de sumas simples. Este ejemplo es similar a encontrar la rapidez promedio de un automóvil que se conduce hacia una ciudad a 100 km de distancia a 100 km/h y en el viaje de retorno a 50 km/h; la rapidez promedio no es $(100 + 50)/2 = 75$ km/h, sino que se debe obtener del hecho de que el automóvil recorre 200 km en tres horas.

Puesto que los tiempos de ejecución de los *benchmarks* SPEC se normalizan a una máquina de referencia más que expresarse en términos absolutos, se debe buscar también resumir en este caso datos de rendimiento. Por ejemplo, si en la tabla 4.4 se toma X como la máquina de referencia, entonces el rendimiento normalizado de Y está dado por los valores de aceleración en la columna de la derecha. Se sabe de la discusión precedente que el promedio (media aritmética) de las aceleraciones (tiempos de ejecución normalizados) no es la medida correcta a usar. Se necesita una forma de resumir tiempos de ejecución normalizados, que sea consistente y su resultado no dependa de la máquina de referencia elegida, X exhibiría aceleraciones de 10, 0.1 y 0.1 para los programas A, B y C, respectivamente. El promedio de estos valores es 3.4; no sólo no se obtuvo el inverso de 6.7 (aceleración pretendida de Y sobre X), ¡sino que se llegó a la conclusión contradictoria de que cada máquina es más rápida que la otra!

Una solución al problema anterior es usar la *media geométrica* en lugar de la media aritmética. La media geométrica de n valores es la raíz n -ésima de su producto. Al aplicar este método a los valores de aceleración en la tabla 4.4, se obtiene el solo indicador de rendimiento relativo de $(0.1 \times 10 \times 10)^{1/3} = 2.15$ para Y relativo a X. Observe que no se llama a esta aceleración global de Y sobre X porque no lo es; es sólo un indicador que se mueve en la dirección correcta en el sentido en que valores más grandes corresponde a mayor rendimiento. De haber usado Y como la máquina de referencia, el indicador de rendimiento relativo para X habría sido $(10 \times 0.1 \times 0.1)^{1/3} = 0.46$. Ahora esto es consistente, porque 0.46 es el inverso aproximado de 2.15. Esta consistencia surge del hecho de que la razón de medias geométricas es la misma que la media geométrica de razones.

Usar la media geométrica resuelve el problema de consistencia pero crea otro problema: los números derivados no tienen relación directa con los tiempos de ejecución y, de hecho, pueden ser bastante engañosos. Considere, por ejemplo, sólo los programas A y B en la tabla 4.4. Con base en estos dos programas, las máquinas X y Y tienen el mismo rendimiento, puesto que $(0.1 \times 10)^{1/2} = 1$. Aunque éste claramente no es el caso si los programas A y B se ejecutan el mismo número de veces en la carga de trabajo. Los tiempos de ejecución en las dos máquinas serían los mismos sólo si la fracción a de ejecuciones que corresponde al programa A (por tanto, $1 - a$ para el programa B) satisface la siguiente igualdad:

$$a \times 20 + (1 - a) \times 1000 = a \times 200 + (1 - a) \times 100$$

Esto requiere $a = 5/6$ y $1 - a = 1/6$, lo que implica que el programa A se debe ejecutar cinco veces lo que el programa B; éste puede o no ser el caso en la carga de trabajo.

Ejemplo 4.6: Efecto de la mezcla de instrucciones en el rendimiento Considere las aplicaciones de compresión de datos y de simulación de reactor nuclear en la tabla 4.3 y suponga que los CPI promedio para las instrucciones de las clases A-F en dos máquinas M_1 y M_2 son:

Clase	CPI promedio para M_1	CPI promedio para M_2	Comentarios
A	4.0	3.8	Load/Store
B	1.5	2.5	Integer arithmetic
C	1.2	1.2	Shift/Logical
D	6.0	2.6	Floating-point
E	2.5	2.2	Branch
F	2.0	2.3	Todos los demás

- Calcule los CPI efectivos para estas dos aplicaciones en cada máquina (cuatro resultados).
- Para cada aplicación, calcule la aceleración de M_2 sobre M_1 , suponga que ambas máquinas tienen la misma tasa de reloj.
- Use media geométrica, cuantifique la ventaja de rendimiento global de M_2 sobre M_1 .

Solución

- CPI para la aplicación de compresión de datos en $M_1 = 0.25 \times 4.0 + 0.32 \times 1.5 + 0.16 \times 1.2 + 0 \times 6.0 + 0.19 \times 2.5 + 0.08 \times 2.0 = 2.31$

CPI para aplicación de compresión de datos en $M_2 = 2.54$

CPI para aplicación de simulación de reactor nuclear en $M_1 = 3.94$

CPI para aplicación de simulación de reactor nuclear en $M_2 = 2.89$
- Dado que los programas de las tasas de reloj son los mismos, la aceleración está dada por la razón de CPI: $2.31/2.54 = 0.91$ para compresión de datos (1.10 frenado), $3.94/2.89 = 1.36$ para simulación de reactor nuclear.
- La ventaja de rendimiento global de M_2 sobre M_1 es $(0.91 \times 1.36)^{1/2} = 1.11$.

4.6 Búsqueda de mayor rendimiento

El estado de potencia computacional disponible a la vuelta del siglo xxi se puede resumir del modo siguiente:

Gigaflops en las computadoras de escritorio.

Teraflops en el centro de supercomputadoras.

Petaflops en la mesa de dibujo.

Dado el crecimiento exponencial en el rendimiento de computadoras, en 10-15 años se atestiguará un cambio de G, T y P hacia T, P y E (tabla 3.1). A través de los años, lograr hitos de rendimiento, como teraflops y petaflops, ha sido una de las principales fuerzas conductoras en la arquitectura y tecnología de computadoras.

Ciertas agencias gubernamentales en Estados Unidos y otros usuarios avanzados apoyan proyectos de investigación y desarrollo en supercomputadoras para ayudar a resolver problemas más grandes, o hasta ahora muy difíciles, dentro de sus dominios de interés. Con el tiempo, nuevos métodos de mejora de rendimiento que se introdujeron en las supercomputadoras de perfil alto encuentran su camino hacia los sistemas más pequeños, y eventualmente se muestran en las computadoras personales. Así, las principales compañías de computadoras también están activas en el diseño y construcción de sistemas

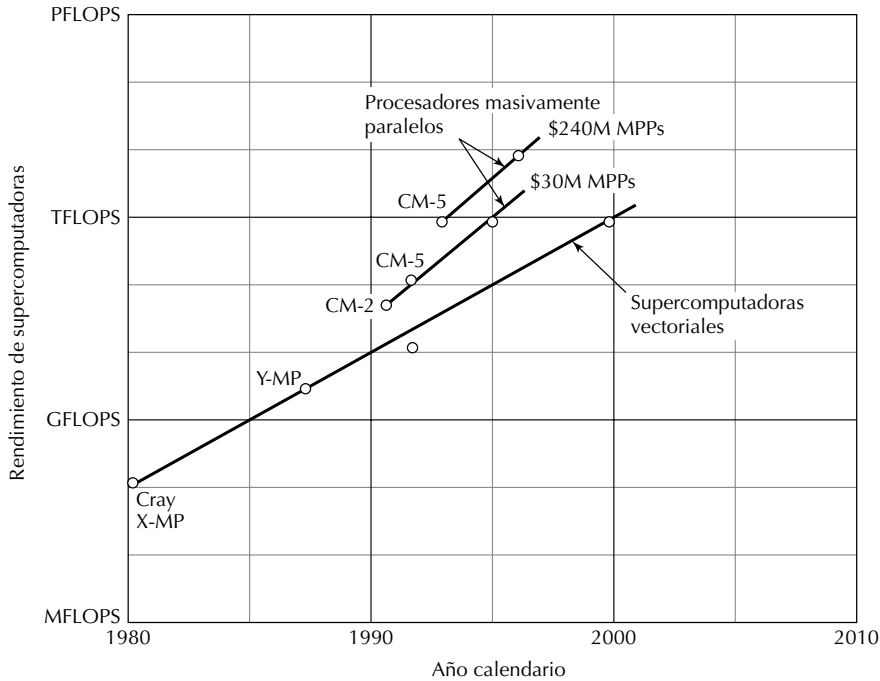


Figura 4.7 Crecimiento exponencial del rendimiento de supercomputadoras [Bell92].

de cómputo de máximo rendimiento, aun cuando el mercado para tales máquinas costosas de perfil alto sea muy limitado (figura 3.4). Como se ve en la figura 4.7, el rendimiento de las supercomputadoras sigue un crecimiento exponencial. Esta tendencia se aplica tanto a las supercomputadoras vectoriales como a los procesadores masivamente paralelos (MPP, por sus siglas en inglés).

Para asegurar que el progreso en el rendimiento de supercomputadoras no se frena por el alto costo de investigación y desarrollo de tales máquinas, el Departamento de Energía de Estados Unidos patrocina el programa Accelerated Strategic Computing Initiative (ASCI, Iniciativa de Computación Estratégica Acelerada), descrito como la Advanced Simulation and Computing Initiative (Iniciativa de Simulación y Computación Avanzada), que tiene como meta el desarrollo de supercomputadoras que marquen límites del rendimiento: de 1 TFLOPS en 1997 a 100 TFLOPS en 2003 (figura 4.8). Aun cuando estos números corresponden a rendimiento pico, existe esperanza de que los aumentos en la potencia computacional pico significarán avances impresionantes en rendimiento sostenido para aplicaciones reales.

En lo futuro se espera que el rendimiento tanto de microprocesadores como de supercomputadoras crezca al ritmo actual. De este modo, la interpolación de las tendencias que se aprecian en las figuras 4.7 y 4.8 conduce a predicciones precisas del nivel de rendimiento que se logrará en la década siguiente. Sin embargo, más allá de ello, el panorama es mucho menos claro. El problema es que se está llegando a ciertos límites físicos fundamentales que puede ser difícil, o incluso imposible, de superar. Una preocupación es que la reducción del tamaño característico de los circuitos integrados, que es un importante contribuyente a las mejoras en rapidez, se está acercando cada vez más a dimensiones atómicas. Otro conflicto es que la rapidez de la propagación de señal en los conectores entre elementos de chip es inherentemente limitada; en la actualidad es una fracción de la rapidez de la luz y nunca podrá superar la última (aproximadamente 30 cm/ns). Por ejemplo, si un chip de memoria está a 3 cm del chip procesador, nunca se podrá esperar enviar datos de uno al otro en un tiempo más corto que 0.1 ns. En consecuencia, es necesario más trabajo

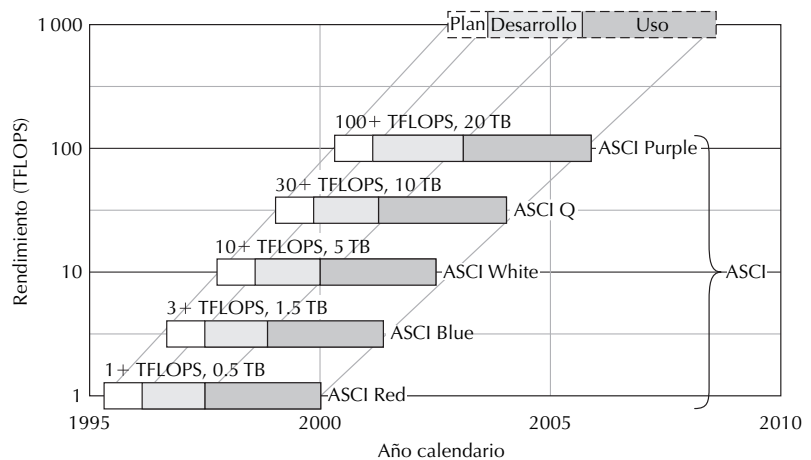


Figura 4.8 Hitos en el programa Accelerated Strategic Computing Initiative (ASCI), patrocinado por el Departamento de Energía de Estados Unidos, con extrapolación a nivel de PFLOPS.

en técnicas arquitectónicas que obvian la necesidad de frecuente comunicación a larga distancia entre elementos de circuito. Observe que el procesamiento paralelo por sí mismo no resuelve el dilema de la “velocidad de la luz”; los procesadores múltiples todavía necesitan comunicarse entre ellos.

PROBLEMAS

4.1 Ley de Amdahl

Un programa corre a mil segundos en una máquina particular, con 75% del tiempo empleado en la realización de operaciones multiplicar/dividir. Se quiere rediseñar la máquina para dotarla con hardware multiplicar/dividir más rápido.

- ¿Cuánto más rápido se volvería el multiplicador/divisor para que el programa corra tres veces con mayor rapidez?
- ¿Y si se quiere que el programa corra cuatro veces más veloz?

4.2 Benchmarks de rendimiento

Una suite *benchmark* B_1 consta de igual proporción de instrucciones de clase X y de clase Y. Las máquinas M_1 y M_2 , con idénticos relojes de 1 GHz, tienen igual rendimiento de 500 MIPS en B_1 . Si se sustituye la mitad de las instrucciones de clase X en B_1 con instrucciones de clase Y para derivar otra suite *benchmark* B_2 , el tiempo de corrido de M_1 se vuelve 70% que el de M_2 . Si sus-

tituimos la mitad de las instrucciones de clase Y con instrucciones de clase X para transformar B_1 en B_3 , M_2 se vuelve 1.5 veces más rápido que M_1 .

- ¿Qué puedes decir acerca de los CPI promedio de M_1 y M_2 para las dos clases de instrucción?
- ¿Cuál es el rendimiento IPS de M_1 en las suites *benchmark* B_2 y B_3 ?
- ¿Cuál es la máxima aceleración posible de M_1 sobre M_2 y para qué mezcla de instrucciones se logra?
- Repita la parte c) para el rendimiento de M_2 relativa a M_1 .

4.3 Ciclos por instrucción

En los ejemplos de este capítulo, los CPI proporcionados y calculados son mayores que o igual a 1. ¿CPI puede ser menor que 1? Explique.

4.4 Frecuencia de reloj y aceleración

Una compañía vende dos versiones de sus procesadores. La versión “pro” corre a 1.5 veces la frecuencia de

reloj pero tiene un CPI promedio de 1.50 frente a 1.25 para la versión “deluxe”. ¿Qué aceleración, en relación con la versión deluxe, esperarías cuando corre un programa en la versión pro?

4.5 Comparación y aceleración de rendimiento

Considere dos diferentes implementaciones M_1 y M_2 del mismo conjunto de instrucciones. M_1 tiene una frecuencia de reloj de 500 MHz. El ciclo de reloj de M_2 es de 1.5 ns. Existen tres clases de instrucciones con los siguientes CPI:

Clase	CPI para M_1	CPI para M_2
A	2	2
B	1	2
C	3	4

- ¿Cuáles son los rendimientos pico de M_1 y M_2 , expresados como MIPS?
- Si el número de instrucciones ejecutadas en cierto programa se divide igualmente entre las tres clases, ¿cuál máquina es más rápida y en qué factor?
- Se puede rediseñar M_2 tal que, con aumento de costo despreciable, los CPI para las instrucciones de la clase B mejore de 2 a 1 (los CPI de las clases A y C permanecen invariados). Sin embargo, este cambio aumentaría el ciclo de reloj de 1.5 ns a 2 ns. ¿Cuál debería ser el porcentaje mínimo de instrucciones de clase B en una mezcla de instrucciones para que este rediseño resulte en rendimiento mejorado?

4.6 Ley de Amdahl

Un programa emplea 60% de su tiempo de ejecución realizando aritmética de punto flotante. De las operaciones de punto flotante en este programa, 90% se ejecutan en ciclos paralelizables.

- Encuentre la mejora en tiempo de ejecución si el hardware de punto flotante se hace el doble de rápido.
- Encuentre la mejora en tiempo de ejecución si se usan dos procesadores para correr los ciclos paralelizables del programa el doble de rápido.
- Encuentre la mejora en el tiempo de ejecución que resulta de modificaciones tanto en a) como en b).

4.7 Mezcla de instrucciones y rendimiento

Este problema es una continuación del ejemplo 4.6. La máquina M_1 se puede rediseñar de modo que su tasa

de reloj sea 1.4 veces la tasa actual (es decir, 1.4 GHz en lugar de 1 GHz). Hacer esto último requerirá otros cambios de diseño que aumenten todos los CPI en 1. ¿Cómo se compara M_2 con M_1 rediseñada en términos de rendimiento?

4.8 Mezcla de instrucciones y rendimiento

Vuelva a hacer el ejemplo 4.6, pero, en lugar de considerar sólo compresión de datos y simulación de reactor nuclear, considere las cuatro aplicaciones que se mencionan en la tabla 4.3.

- Tabule los resultados de las aplicaciones individuales en cada máquina.
- Encuentre resultados compuestos para las dos aplicaciones enteras (compresión de datos y compilador lenguaje C) y dos puntos flotantes (simulación de reactor nuclear y modelado de movimiento atómico) al promediar las mezclas de instrucciones.
- Al usar media geométrica y los resultados de la parte b), cuantifique la ventaja de rendimiento global de M_2 sobre M_1 .

4.9 Ganancia de rendimiento con procesamiento vectorial

Una supercomputadora vectorial tiene instrucciones especiales que realizan operaciones aritméticas sobre vectores. Por ejemplo, la multiplicación vectorial sobre los vectores A y B de longitud 64 es equivalente a 64 multiplicaciones independientes $A[i] \times B[i]$. Suponga que la máquina tiene un CPI de 2 en todas las instrucciones aritméticas escalares. La aritmética vectorial sobre vectores de longitud m toma $8 + m$ ciclos, donde 8 es la cabecera *star-up/wind-down* (arranque/reducción) para el encauzamiento que permite que una operación aritmética se inicia en cada ciclo de reloj; por ende, la multiplicación vectorial toma 72 ciclos de reloj para vectores de longitud 64. Considere un programa sólo con instrucciones aritméticas (es decir, ignore todo lo demás), con la mitad de estas instrucciones involucrando escalares y la mitad involucrando operandos vectoriales.

- ¿Cuál es la aceleración que se logra si la longitud vectorial promedio es 16?
- ¿Cuál es la longitud vectorial de equilibrio para esta máquina (longitud vectorial promedio para resultar en igual o mayor rendimiento debido a procesamiento vectorial)?

- c) ¿Cuál es la longitud vectorial promedio que se requiere para lograr una aceleración de 1.8?

4.10 Ley de Amdahl

Una nueva versión de una máquina M, llamada Mfp++, ejecuta todas las instrucciones de punto flotante cuatro veces más rápido que M.

- Grafique la aceleración que logra la Mfp++ relativa a M como función de la fracción x de tiempo que M gasta en aritmética de punto flotante.
- Encuentre la aceleración de Mfp++ sobre M para cada una de las aplicaciones que se muestran en la tabla 4.3, si supone que los CPI promedio para instrucciones de punto flotante en M es cinco veces la de todas las otras instrucciones.

4.11 Calificación MIPS

Una máquina que usa su compañía tiene un CPI promedio de cuatro para instrucciones aritméticas de punto flotante y uno para todas las otras instrucciones. Las aplicaciones que usted corre gastan la mitad de su tiempo en aritmética de punto flotante.

- ¿Cuál es la mezcla de instrucciones en sus aplicaciones? Esto es, encuentre la fracción x de instrucciones ejecutadas que realizan aritmética de punto flotante.
- ¿Cuánto más alta sería la calificación MIPS de la máquina si usara un compilador que simulase aritmética de punto flotante mediante secuencias de instrucciones enteras en lugar de usar instrucciones de punto flotante de la máquina?

4.12 Comparación y aceleración de rendimiento

Considere dos implementaciones diferentes M_1 (1 GHz) y M_2 (1.5 GHz) del mismo conjunto de instrucciones. Existen tres clases de instrucciones con los siguientes CPI:

Clase	CPI para M_1	CPI para M_2
A	2	2
B	1	2
C	3	5

- ¿Cuáles son los rendimientos pico de M_1 y M_2 expresados como MIPS?
- Demuestre que si la mitad de las instrucciones ejecutadas en cierto programa son de la clase A y el

resto se dividen igualmente entre las clases B y C, entonces M_2 es más rápido que M_1 .

- Demuestre que la segunda suposición de la parte b) es redundante. En otras palabras, si la mitad de las instrucciones ejecutadas en cierto programa son de la clase A, entonces M_2 siempre será más rápido que M_1 , sin importar la distribución de las instrucciones restantes entre las clases B y C.

4.13 Tendencias de rendimiento de supercomputadoras

Los datos acerca de las supercomputadoras más potentes en la actualidad se publican con regularidad [Top500]. Dibuje las siguientes gráficas de dispersión con base en los datos para las cinco mejores supercomputadoras en cada uno de los últimos diez años y discuta las tendencias observadas:

- Rendimiento frente a número de procesadores.
- Rendimiento frente a cantidad total de memoria y memoria por procesador (use dos tipos de marcador en la misma gráfica de dispersión)
- Número de procesadores frente a año de introducción

4.14 Rendimiento relativo de máquinas

Al referirse a la figura 4.5, se nota que el rendimiento relativo de las máquinas 1 y 2 fluctúa ampliamente para los seis programas mostrados. Por ejemplo, la máquina 1 es el doble de rápida para el programa B, mientras que lo opuesto es cierto para el programa A. Especule acerca de qué diferencias en las dos máquinas y los seis programas pueden haber contribuido a los tiempos de ejecución observados. Entonces, compare la máquina 3 con cada una de las otras dos máquinas.

4.15 Ley de Amdahl

Usted vive en un departamento desde el cual tiene que conducir siete minutos para su viaje de compras dos veces a la semana hasta un supermercado cercano, y un viaje de 20 minutos hacia una tienda departamental donde compra una vez cada cuatro semanas. Usted planea mudarse a un nuevo departamento. Compare las siguientes ubicaciones posibles con respecto a la aceleración que le ofrecen para su tiempo de conducción durante los viajes de compras:

- a) Un departamento que está a diez minutos de distancia del supermercado y de la tienda departamental.
- b) Un departamento que está a cinco minutos de distancia del supermercado y a 30 minutos de la tienda departamental.

4.16 Ley de Amdahl

Suponga que, con base en cuentas de operación (no hay gasto de tiempo en ellas), una aplicación numérica usa 20% de operaciones de punto flotante y 80% de operaciones *integer*/control (entero/control). El tiempo de ejecución de una operación de punto flotante es, en promedio, tres veces más larga que otras operaciones. Se considera un rediseño de la unidad de punto flotante para hacerla más rápida.

- a) ¿Qué factor de aceleración para la unidad de punto flotante conduciría a 25% de mejora global en rapidez?
- b) ¿Cuál es la máxima aceleración posible que se puede lograr al modificar sólo la unidad de punto flotante?

4.17 Benchmark de rendimiento y MIPS

Se pretende que un *benchmark* particular se ejecute repetidamente, con el rendimiento computacional especificado como el número de veces que el *benchmark* se puede ejecutar por segundo. De este modo, una tasa de repetición de 100/s representa un mayor rendimiento que 80/s. Las máquinas M_1 y M_2 exhiben un rendimiento de R_1 y R_2 repeticiones por segundo. Durante estas ejecuciones, M_1 tiene un rendimiento MIPS que se califica como P_1 . ¿Sería correcto concluir que la calificación MIPS de M_2 en este *benchmark* es $P_1 \times R_2/R_1$? Justifique su respuesta completamente.

4.18 Analogía con rendimiento de aeronaves

En la sección 4.1 y en la tabla 4.1 se compararon varias aeronaves de pasajeros con respecto a su rendimiento y se observó que el rendimiento se juzga de manera diferente dependiendo del punto de vista del evaluador. La información acerca de aeronaves militares estadounidenses está disponible en los sitios Web de Periscope (periscopeone.com), NASA y el American Institute of Aeronautics and Astronautics.

- a) Construya una tabla similar a la 4.1, mencione parámetros relevantes relacionados con el rendimiento

de jets de combate estadounidenses (F-14, F-15/15E, F-16 y F/A-18) y discuta cómo éstos se comparan desde distintos puntos de vista.

- b) Repita la parte a) para bombarderos del ejército estadounidense, como los B-52-H, B-1B, B-2 y F-117.
- c) Repita la parte a) para vehículos aéreos disuasivos del ejército estadounidense como Black Widow, Hunter, Predator y Global Hawk.

4.19 Tendencias de supercomputadoras

La *Science and Technology Review* editó una cronología de supercomputadoras en el Lawrence Livermore National Laboratory, desde la UNIVAC 1, en 1953, hasta la ASCII White de diez teraoperaciones, en 2000 [Park02].

- a) A partir de estos datos, derive una cifra de tasa de crecimiento para el rendimiento de supercomputadoras a lo largo de las últimas cinco décadas del siglo xx y compárela contra el crecimiento de rendimiento de microprocesadores en la figura 3.10.
- b) ¿La tasa de crecimiento de la parte a) es consistente con la figura 4.7? Discuta.
- c) Intente obtener datos acerca del tamaño de memoria principal en estas computadoras. Derive tasas de crecimiento para tamaño de memoria principal y compárela con la tasa de crecimiento de rendimiento de la parte a).
- d) Repita la parte c) para memoria masiva.

4.20 Calificación MIPS

Una computadora tiene dos clases de instrucciones. Las instrucciones de clase S tienen un CPI de 1.5 y las de clase C tienen un CPI de 5.0. La tasa de reloj es de 2 GHz. Sea x la fracción de instrucciones en un conjunto de programas de interés que pertenecen a la clase S.

- a) Grafique la variación en calificación MIPS para la máquina conforme x varía de 0 a 1.
- b) Especifique el rango de valores x para los que una aceleración de 1.2 para las instrucciones de la clase S conduce a mejor rendimiento que una aceleración de 2.5 para las instrucciones de la clase C.
- c) ¿Cuál sería una calificación promedio MIPS justa para esta máquina si no se sabe nada acerca de x ? (Es decir, el valor de x se distribuye uniformemente sobre $[0, 1]$ para diferentes aplicaciones de interés potencial.)

REFERENCIAS Y LECTURAS SUGERIDAS

- [Bell92] Bell, G., “Ultracomputers: ATeraflop Before Its Time”, *Communications of the ACM*, vol. 35, núm. 8, pp. 27-47, agosto de 1992.
- [Crow94] Crowl, L. A., “How to Measure, Present, and Compare Parallel Performance”, *IEEE Parallel & Distributed Technology*, vol. 2, núm. 1, pp. 9-25, primavera de 1994.
- [Henn00] Henning, J. L., “SPEC CPU2000: Measuring CPU Performance in the New Millennium”, *IEEE Computer*, vol. 33, núm. 7, pp. 28-35, julio de 2000.
- [Henn03] Hennessy, J. L. y D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 3a. ed., 2003.
- [Park02] Parker, A., “From Kilobytes to Petabytes in 50 Years”, *Science and Technology Review*, publicado por LLNL, pp. 20-26, marzo de 2002. <http://www.llnl.gov/str/>
- [Patt98] Patterson, D. A. y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2a. ed., 1998.
- [Smit88] Smith, J. E., “Characterizing Computer Performance with a Single Number”, *Communications of the ACM*, vol. 31, núm. 10, pp. 1202-1206, octubre de 1988.
- [Top500] Sitio de las 500 mejores supercomputadoras, sitio Web mantenido por la Universidad de Mannheim y la Universidad de Tennessee: <http://www.top500.org/>

PARTE DOS

ARQUITECTURA DE CONJUNTO DE INSTRUCCIONES

“Los antropólogos del siglo xxi quitarán capa tras capa de tales [poco elegantes, fortuitamente extensas] máquinas hasta que descubran artefactos del primer microprocesador. Ya tal hallazgo, ¿cómo juzgarán la arquitectura de computadoras del siglo xx ?”

David A. Patterson y John L. Hennessy, Computer Organization and Design, 1998

“En todas las matemáticas que he realizado, el punto esencial fue encontrar la arquitectura correcta. Es como construir un puente. Cuando las principales líneas de la estructura son correctas, los detalles encajan milagrosamente.”

Freeman Dyson, entrevista, 1994

TEMAS DE ESTA PARTE

5. Instrucciones y direccionamiento
6. Procedimientos y datos
7. Programas en lenguaje ensamblador
8. Variaciones en el conjunto de instrucciones

Las instrucciones son palabras del lenguaje que entiende la máquina, el conjunto de instrucciones es su vocabulario. Un programador que escribe programas para la máquina, algo extraordinario en estos días, o un compilador que traslada programas en lenguaje de alto nivel al lenguaje máquina, deben entender este vocabulario y dónde se puede usar correctamente cada palabra. La arquitectura del conjunto de instrucciones de una máquina es su vocabulario junto con partes de la máquina y sus funciones, que debe dominar un usuario para producir programas correctos, compactos y rápidos.

En los capítulos 5 y 6 se introduce el núcleo de una simple, aunque realista y útil, arquitectura de conjunto de instrucciones; partes adicionales del conjunto de instrucciones se introducirán más adelante en el libro, en particular en la parte cuatro, que trata de la ALU. La estructura de los programas en lenguaje ensamblador se discute en el capítulo 7. A lo largo de todo este camino, el énfasis no es dominar este conjunto de instrucciones particular o ser capaz de programar en él sin esfuerzo; más bien, la meta es aprender lo suficiente acerca de este conjunto de instrucciones para comprender las razones detrás de las diferencias entre las máquinas y cómo diversas opciones afectan el costo del hardware y su rendimiento. En el capítulo 8 se cubren la dicotomía RISC/CISC y otros aspectos del diseño del conjunto de instrucciones.

■ CAPÍTULO 5

INSTRUCCIONES Y DIRECCIONAMIENTO

“Aquí todavía tenemos el juicio de que enseñamos malditas lecciones, que, al enseñarse, regresan para maldecir al inventor.”

William Shakespeare, Macbeth

“Lo más importante en el lenguaje de programación es el nombre. Un lenguaje no triunfará sin un buen nombre. Hace poco inventé un nombre muy bueno y ahora busco un lenguaje adecuado.”

Donald E. Knuth, 1967

TEMAS DEL CAPÍTULO

- 5.1 Visión abstracta del hardware
- 5.2 Formatos de instrucción
- 5.3 Aritmética simple e instrucciones lógicas
- 5.4 Instrucciones *load* y *store*
- 5.5 Instrucciones *jump* y *branch*
- 5.6 Modos de direccionamiento

En este capítulo comienza el estudio de un conjunto de instrucciones simple que ayudará a comprender los elementos de una moderna arquitectura de conjunto de instrucciones, las opciones involucradas en su elaboración y aspectos de su ejecución en hardware. El conjunto de instrucciones elegido se llama MiniMIPS; éste define una máquina hipotética que está muy cerca de los procesadores MIPS reales, un conjunto de procesadores que ofrece una compañía del mismo nombre. Este conjunto de instrucciones está muy bien documentado, se ha usado en muchos otros libros de texto y tiene un simulador gratuito que se puede descargar para ejercicios de práctica y programación. Al final de este capítulo será capaz de componer secuencias de instrucciones para realizar tareas computacionales no triviales.

■ 5.1 Visión abstracta del hardware

Para conducir un automóvil, se debe familiarizar con algunos elementos clave como los pedales de acelerador y freno, el volante y algunos instrumentos del tablero. De manera colectiva, estos dispositivos le permiten controlar el automóvil y sus diversas partes, así como observar el estado de ciertos subsistemas cruciales. La interfaz correspondiente para el hardware de computadora es su arquitectura de conjunto de

instrucciones. Es necesario aprender esta interfaz para ser capaz de instruir a la computadora a realizar tareas computacionales de interés. En el caso de los automóviles, la interfaz del usuario es tan estandarizada que fácilmente uno puede operar un automóvil rentado de una marca que nunca antes se haya conducido. No se puede decir lo mismo acerca de las computadoras digitales, aunque a lo largo del tiempo se han desarrollado muchas características de conjunto de instrucciones comunes. Cuando se familiarice con el conjunto de instrucciones de una máquina, aprenderá otros con poco esfuerzo; el proceso es más parecido a mejorar el vocabulario o estudiar un nuevo dialecto del español que aprender todo un nuevo idioma.

El conjunto de instrucciones MiniMIPS (MIPS mínimos) es bastante similar a lo que uno encuentra en muchos procesadores modernos. MiniMIPS es un conjunto de instrucciones *load/store* (carga/almacenamiento), ello significa que los elementos de datos se deben copiar o *load* (cargar) en registros antes de procesarlos; los resultados de operación también van hacia registros y se deben copiar explícitamente de vuelta a la memoria a través de operaciones *store* (almacenamiento) separadas. Por ende, para comprender y ser capaz de usar MiniMIPS, es necesario saber acerca de esquemas de almacenamiento de datos en memoria, funciones de las instrucciones de carga y almacenamiento, tipos de operaciones permitidas en los elementos de datos que se conservan en los registros y algunos otros aspectos sueltos que permiten la programación eficiente.

Para conducir un automóvil no se necesita saber dónde se ubica el motor o cómo acciona éste las ruedas; sin embargo, la mayoría de los instructores comienzan a enseñar la conducción al mostrar a sus estudiantes un diagrama de un automóvil con sus diversas partes. En esta coyuntura, la figura 5.1 se presenta con el mismo espíritu, aunque en partes ulteriores del libro se examinará el hardware de computadora con mucho más detalle. El programador (compilador) de lenguaje ensamblador se relaciona con:

- Registros.
- Localidades (*locations*) de memoria donde se pueden almacenar datos.
- Instrucciones de máquina que operan sobre y almacenan datos en los registros o memoria.

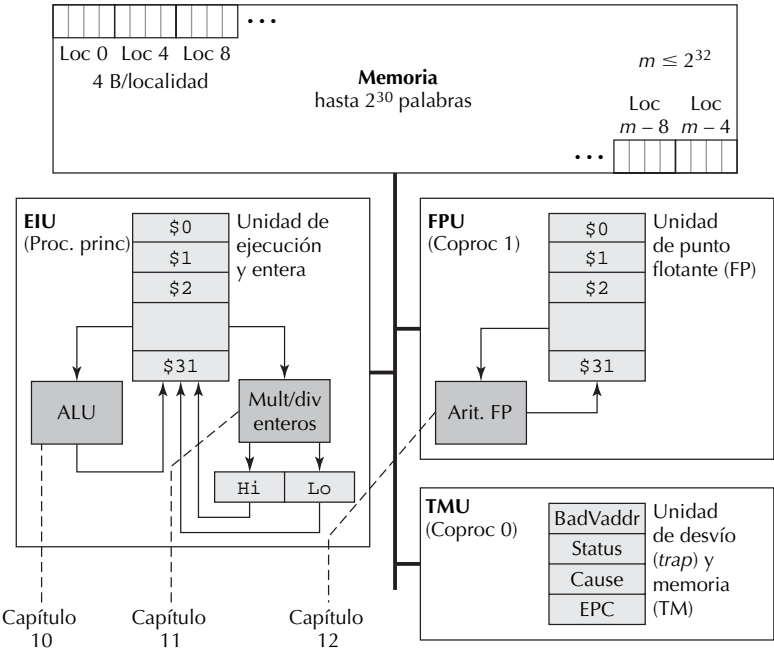


Figura 5.1 Subsistemas de memoria y procesamiento de los MiniMIPS.

La figura 5.1 muestra la unidad de memoria MiniMIPS, con hasta 2^{30} palabras (2^{32} bytes), una unidad de ejecución y entero (EIU, por sus siglas en inglés), una unidad de punto flotante (FPU, por sus siglas en inglés) y una unidad de desvío y memoria (TMU, por sus siglas en inglés). FPU y TMU se muestran por completo; en esta parte del libro no se examinarán las instrucciones que ejercen estas unidades. Las instrucciones relacionadas con FPU se tratarán en el capítulo 12, mientras que en la parte seis se revelarán algunos de los usos de la TMU. La EIU interpreta y ejecuta las instrucciones MiniMIPS básicas que se cubren en esta parte del libro, y tiene 32 registros de propósito general, cada uno con 32 bits de ancho; por ende, puede retener el contenido de una ubicación de memoria. La unidad aritmética/lógica (ALU, por sus siglas en inglés) ejecuta las instrucciones *addition* (suma), *subtraction* (resta) y *logical* (lógica). Una unidad aritmética separada se dedica a las instrucciones *mutiplicación* y *división*, cuyos resultados se colocan en dos registros especiales, llamados “Hi” y “Lo”, desde donde se pueden mover hacia los registros de propósito general.

En la figura 5.2 se muestra una visión de los registros MiniMIPS. Todos los registros, excepto el 0 (\$0), que retiene permanentemente la constante 0, son de propósito general y se pueden usar para almacenar

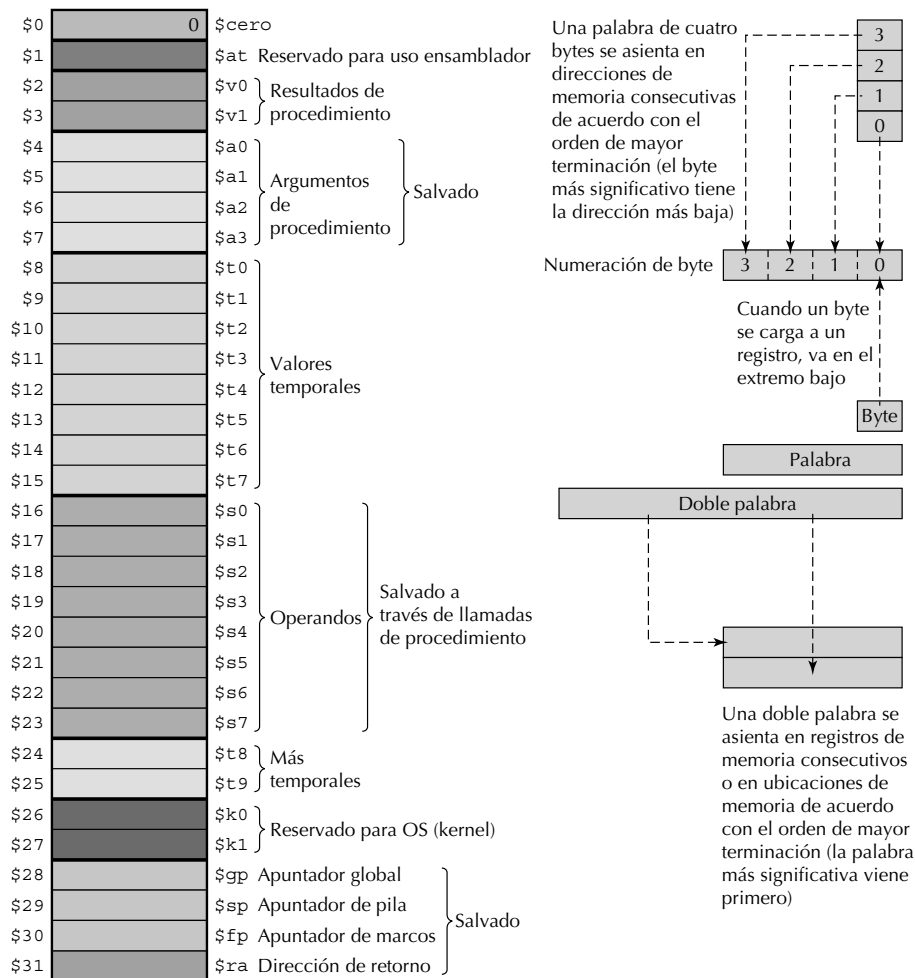


Figura 5.2 Registros y tamaños de datos en MiniMIPS.

palabras de datos arbitrarias. Sin embargo, para facilitar el desarrollo y compilación eficientes de programa, son usuales ciertas restricciones en el uso de registros. Una analogía útil para estas restricciones es la forma en que una persona puede decidir poner cambio, llaves, cartera, etc., en bolsillos específicos para hacer más fácil recordar dónde se pueden encontrar los artículos. Las restricciones que se usan en este libro también se muestran en la figura 5.2. La mayoría de estas convenciones no tienen sentido para usted en este momento. Por tanto, en los ejemplos sólo se usarán los registros del \$8 al \$25 (con nombres simbólicos \$s0–\$s7 y \$t0–\$t9) hasta que se discuta el uso de convenciones para los otros registros.

A un elemento de datos de 32 bits almacenado en un registro o localidad de memoria (con una dirección divisible entre 4) se le conoce como *word* (palabra). Por el momento, suponga que una palabra retiene una instrucción o un entero con signo, aunque más adelante se verá que también puede retener un entero sin signo, un número en punto flotante o una cadena de caracteres ASCII. Puesto que las palabras MiniMIPS se almacenan en una memoria direccionable en bytes, es necesaria una convención para establecer cuál extremo de la palabra aparece en el primer byte (el de dirección de memoria más baja). De las dos posibles convenciones a este respecto, los MiniMIPS usan el esquema *big-endian* (mayor terminación), donde el extremo más significativo aparece primero.

Para ciertos valores que no necesitan todo el rango de una palabra de 32 bits, se pueden usar bytes de ocho bits. Cuando un elemento de dato de tamaño byte se pone en un registro, aparece en el extremo derecho del registro (byte más bajo). Una *doubleword* (doble palabra) ocupa dos registros o localidades de memoria consecutivos. De nuevo, la convención acerca del orden de las dos palabras es *big-endian*. Cuando un par de registros retienen una *doubleword*, el más pequeño de los dos registros siempre tiene un índice par, que se usa para hacer referencia a la localidad *doubleword* (por ejemplo, se dice “la *doubleword* está en el registro \$16” para dar a entender que su extremo alto está en \$16 y el extremo bajo en \$17). De modo que sólo los registros con numeración par pueden retener *doublewords*.

Un punto final antes de enfrascarse en la discusión de instrucciones específicas para MiniMIPS es que, aun cuando un programador (compilador) no necesita saber más acerca de hardware que lo que se cubre en esta parte del libro, los detalles de implementación que se presentan en el resto del libro todavía son bastante importantes. En relación con la analogía del automóvil, usted puede manejar de manera legal prácticamente sin conocimiento de cómo se construye un automóvil o cómo opera su motor. Sin embargo, para aprovechar el máximo rendimiento de la máquina, o para ser un conductor muy seguro, necesita saber mucho más.

■ 5.2 Formatos de instrucción

Una instrucción de máquina MiniMIPS típica es `add $t8, $s2, $s1`, que hace que los contenidos de los registros \$s2 y \$s1 se sumen, y que el resultado se almacene en el registro \$t8. Esto puede corresponder a la forma compilada del enunciado en lenguaje de alto nivel $a = b + c$ y a su vez representa una palabra de máquina que usa 0 y 1 para codificar la operación y especificaciones de registro involucrados (figura 5.3).

Al igual que en el caso de los programas en lenguaje de alto nivel, las secuencias de máquina o instrucciones en lenguaje ensamblador se ejecutan de arriba abajo a menos que se especifique de manera explícita un orden de ejecución diferente mediante las instrucciones *jump* (saltar) o *branch* (bifurcación). Algunas de tales instrucciones se cubrirán en la sección 5.5. Por ahora, enfóquese en el significado de instrucciones simples por sí mismas o en secuencias cortas que correspondan a operación compuesta. Por ejemplo, la siguiente secuencia de instrucciones ensambladas realiza el cálculo $g = (b + c) - (e + f)$ como una secuencia de instrucciones de una sola operación, escritas una por línea. La porción de cada línea que sigue al símbolo “#” es un comentario que se inserta para auxiliar al lector de la secuencia de instrucciones y se ignora durante la ejecución de máquina.

Enunciado en lenguaje de alto nivel:

$a = b + c$

Instrucción en lenguaje ensamblador:

add \$t8, \$s2, \$s1

Instrucción en lenguaje de máquina:

000000	10010	10001	11000	00000	100000
Instrucción tipo ALU	Registro 18	Registro 17	Registro 24	No usado	Código opcode

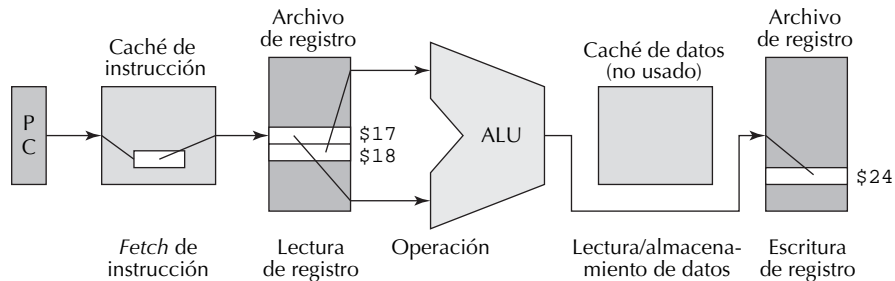


Figura 5.3 Instrucción típica para MiniMIPS y pasos en su ejecución.

```

add      $t8,$s2,$s3      # poner la suma b + c en $t8
add      $t9,$s5,$s6      # poner la suma e + f en $t9
sub      $s7,$t8,$t9      # fija g en ($t8) - ($t9)
  
```

La secuencia de instrucciones precedente calcula g al realizar las operaciones $a = b + c$, $d = e + f$, y $g = a - d$, con valores temporales a y d retenidos en los registros $\$t8$ y $\$t9$, respectivamente.

Al escribir programas en lenguaje de máquina, es necesario proporcionar enteros que especifiquen direcciones de memoria o constantes numéricas. Tales números se especifican en formato *decimal* (base 10) o *hexadecimal* (base 16, o *hexa* para abreviar) y automáticamente se convierten al formato binario para procesamiento de máquina. He aquí algunos ejemplos:

Decimal	25, 123456, -2873
Hexadecimal	0x59, 0x12b4c6, 0xffff0000

Los números hexadecimales se explicarán en la sección 9.2. Por ahora, sólo visualícelos como una notación abreviada para patrones de bits. Los patrones de cuatro bits del 0000 al 1111 corresponden a los 16 dígitos hexadecimales 0–9, a, b, c, d, e y f. De modo que el número hexa 0x59 representa el byte 0101 1001 y 0xffff0000 es abreviatura de la palabra 1111 1111 1111 1111 0000 0000 0000 0000. La razón para usar el prefijo “0x” es distinguir estos números tanto de los números decimales (por ejemplo, 59 o 059 como para nombres de variables (por ejemplo, x59).

Una instrucción de máquina para una operación aritmética/lógica especifica un código de operación (*opcode*), uno o más operandos fuente y, usualmente, un operando de destino. *Opcode* es un código binario (patrón de bits) que define una operación. Los operandos de una instrucción aritmética o lógico pueden provenir de una diversidad de fuentes. Al método que se usa para especificar dónde se encuentran los operandos y a dónde debe ir el resultado se le refiere como modo (o esquema) de

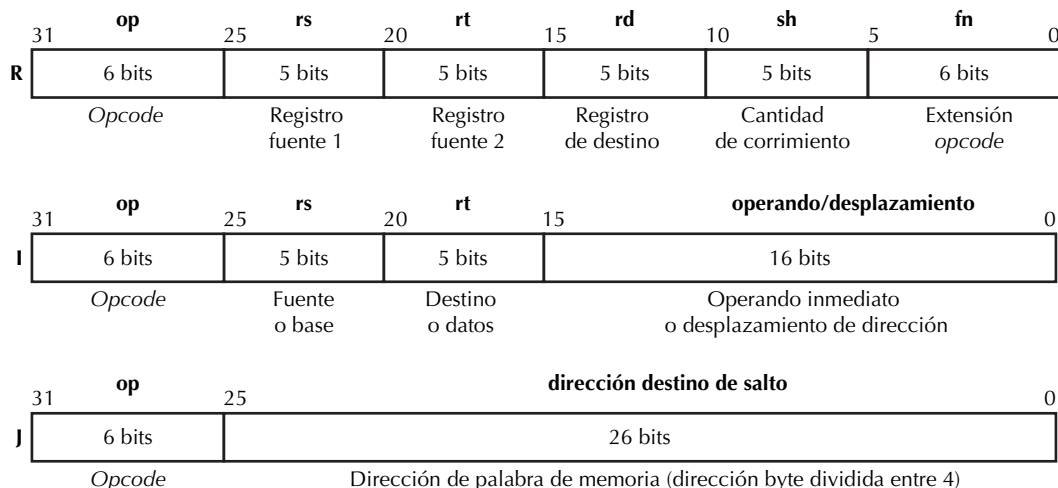


Figura 5.4 Las instrucciones MiniMIPS vienen sólo en tres formatos: registro (R), inmediato (I) y salto (*jump*, J).

direccionamiento. Por el momento, suponga que todos los operandos están en los registros $\$s0-\$s7$ y $\$t0-\$t9$, y otros modos de direccionamiento se discutirán en la sección 5.6.

En la figura 5.4 se muestran los tres formatos de instrucción MiniMIPS. La simplicidad y uniformidad de los formatos de instrucción son comunes en los modernos RISC (*reduced instruction-set computer*, computadora con conjunto de instrucciones reducido) cuya meta es ejecutar las operaciones de uso más común tan rápido como sea posible, acaso a costa de las menos comunes. En el capítulo 8 se discutirán otros enfoques al diseño de conjunto de instrucciones, así como los pro y los contra del enfoque RISC. El campo código de operación (*opcode*), que es común a los tres formatos, se usa para distinguir las instrucciones en las tres clases. De los 64 *opcodes* posibles, los subconjuntos no traslapantes se asignan a las clases de instrucción en forma tal que el hardware puede reconocer fácilmente la clase de una instrucción particular y, en consecuencia, la interpretación adecuada para los campos restantes.

Las instrucciones *register* (registro) o tipo R operan en los dos registros identificados en los campos *rs* y *rt* y almacenan el resultado en el registro *rd*. Para tales instrucciones, el campo función (*fn*) sirve como una extensión de los *opcode*, para permitir que se definan más operaciones y que se use el campo cantidad de corrimiento (*sh*) en instrucciones que especifican una cantidad de corrimiento constante. Las instrucciones de corrimiento (lógico) simple se cubre en el capítulo 6; los corrimientos aritméticos se discutirán en el capítulo 10.

Las instrucciones inmediato (*immediate*) o tipo I en realidad son de dos variedades diferentes. En las instrucciones *immediato*, el campo operando de 16 bits en los bits 0-15 retiene un entero que juega el mismo papel que *rt* en las instrucciones de tipo R; en otras palabras, la operación especificada se realiza en el contenido del registro *rs* y el operando inmediato, el resultado se almacena en el registro *rt*. En las instrucciones *load*, *store* o *branch*, el campo de 16 bits se interpreta como un *offset* (desplazamiento), o dirección relativa, que se debe agregar al valor *base* en el registro *rs* (contador de programa) para obtener una dirección de memoria para lectura o escritura (transferencia de control). Para accesos de datos, el *offset* se interpreta como el número de bytes hacia adelante (positivo) o hacia atrás (negativo) en relación con la dirección base. Para la instrucción *branch*, el *offset* está en palabras, dado que las instrucciones siempre ocupan palabras de memoria completa de 32 bits.

Las instrucciones *jump* (salto) o de tipo J provocan transferencia de control incondicional a la instrucción en la dirección especificada. Puesto que las direcciones MiniMIPS tienen 32 bits de ancho, mientras

que en el campo dirección (*address*) de una instrucción de tipo J sólo están disponibles 26 bits, se usan dos convenciones. Primera, se supone que el campo de 16 bits porta una dirección de palabra en oposición a una dirección de byte; por tanto, el hardware liga dos 0 al extremo derecho del campo *address* de 26 bits para derivar una dirección de palabra de 28 bits. Los cuatro bits aún perdidos se ligan al extremo izquierdo de la dirección de 28 bits en una forma que se describirá en la sección 5.5.

5.3 Aritmética simple e instrucciones lógicas

En la sección 5.2 se introdujeron las instrucciones *add* y *subtract*. Dichas instrucciones trabajan en los registros que contienen palabras completas (32 bits). Por ejemplo:

```
add    $t0,$s0,$s1    # fija $t0 en ($s0)+($s1)
sub    $t0,$s0,$s1    # fija $t0 en ($s0)-($s1)
```

La figura 5.5 bosqueja las representaciones de máquina de las instrucciones precedentes.

Las instrucciones lógicas operan en un par de operandos sobre una base bit a bit. Las instrucciones lógicas en MiniMIPS incluyen las siguientes:

```
and    $t0,$s0,$s1    # fija $t0 en ($s0) ^ ($s1)
or     $t0,$s0,$s1    # fija $t0 en ($s0) v ($s1)
xor    $t0,$s0,$s1    # fija $t0 en ($s0) ⊕ ($s1)
nor    $t0,$s0,$s1    # fija $t0 en ((($s0) v ($s1)) ')
```

Las representaciones de máquina para estas instrucciones lógicas son similares a las de la figura 5.5, pero el campo función retiene 36 (100100) para *and*, 37 (100101) para *or*, 38 (100110) para *xor* y 39 (100111) para *nor*.

Usualmente, un operando de una operación aritmética o lógica es constante. Aunque es posible colocar esta constante en un registro y luego realizar la operación deseada en dos registros, sería más eficiente usar instrucciones que especificaran de manera directa la constante deseada en una instrucción de formato I. Desde luego, la constante debe ser pequeña como para que encaje en el campo de 16 bits que está disponible para este propósito. Por tanto, para valores enteros con signo, el rango válido es de -32768 a 32767 , mientras que para las constantes hexa cualquier número de cuatro dígitos en el rango $[0x000, 0xffff]$ es aceptable.

```
addi    $t0,$s0,61    # fija $t0 en ($s0)+61 (decimal)
```

La representación de máquina para *addi* se muestra en la figura 5.6. No hay instrucción *subtract immediate* correspondiente, dado que su efecto se puede lograr al añadir un valor negativo. Puesto que los MiniMIPS tienen un sumador de 32 bits, el operando inmediato de 16 bits se debe convertir a un valor equivalente de 32 bits antes de aplicarlo como entrada al sumador. Más tarde, en el capítulo 9, se verá que un número signado que se representa en formato de complemento a 2 debe extender su signo



Figura 5.5 Las instrucciones aritméticas *add* y *sub* tienen un formato que es común a todas las instrucciones ALU de dos operandos. Por esto, el campo *fn* especifica la operación aritmética/lógica a realizar.

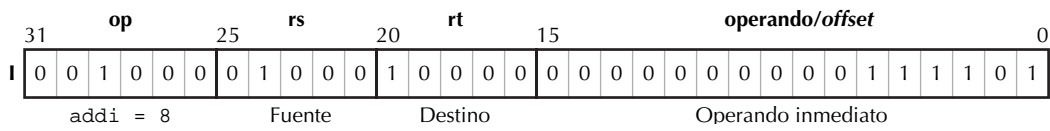


Figura 5.6 Instrucciones como `addi` permiten realizar una operación aritmética o lógica para la que un operando representa una constante pequeña.

si desea resultar en el mismo valor numérico en un formato más ancho. *Extensión de signo* significa simplemente que el bit más significativo de un valor signado de 16 bits se repite 16 veces para llenar la mitad superior de la correspondiente versión de 32 bits. De este modo, un número positivo se extiende con varios 0 mientras que un número negativo se extiende con varios 1.

Tres de las instrucciones lógicas recién introducidas también tienen versiones inmediatas o de formato I:

```
andi    $t0,$s0,61      # fija $t0 en ($s0) ^61
ori     $t0,$s0,61      # fija $t0 en ($s0) v61
xori    $t0,$s0,0x00ff  # fija $t0 en ($s0) ⊕ 0x00ff
```

Las representaciones de máquina para estas instrucciones lógicas son similares a las de la figura 5.6, pero el campo *opcode* retiene 12 (001100) para `andi`, 13 (001101) para `ori` y 14 (001110) para `xori`.

Una diferencia clave entre las instrucciones `andi`, `ori` y `xori` y la instrucción `addi` es que el operando de 16 bits de una instrucción lógica es extendida 0 desde la izquierda para convertirla en formato de 32 bits para procesamiento. En otras palabras, la mitad superior de la versión de 32 bits del operando inmediato para instrucciones lógicas consta de 16 ceros.

Ejemplo 5.1: Extracción de campos de una palabra Una palabra de 32 bits en `$s0` retiene un byte de datos en las posiciones de bit 0-7 y una bandera de estado (*status flag*) en la posición de bit 10. Otros bits en la palabra tienen valores arbitrarios (impredecibles). Desempaque la información en esta palabra, coloque el byte de datos en el registro `$t0` y la bandera de estado en el registro `$t1`. Al final, el registro `$t0` debe retener un entero en [0, 255] correspondiente al byte de datos y el registro `$t1` debe retener un valor distinto de cero si la bandera de estado es 1.

Solución: Los campos de una palabra se pueden extraer al operar AND la palabra con una *máscara* (*mask*) predefinida, una palabra que tenga 1 en las posiciones de bit de interés y 0 en cualquier otra parte. Por ejemplo, al operar AND la palabra con 0x000000ff (que en binario tiene 1 sólo en sus ocho posiciones de bit menos significativas) tiene el efecto de extraer el byte de la extrema derecha. Cualquier bit de bandera deseado se puede extraer de forma similar a través de operar AND con una máscara que tenga un solo 1 en la posición deseada. Por ende, las siguientes dos instrucciones logran lo que se requiere:

```
andi    $t0,$s0,0 00ff  # máscara con ocho 1 en el extremo derecho
andi    $t1,$s0,0 0400  # máscara con un solo 1 en la posición 10
```

Note que el valor distinto de cero que queda en `$t1` en caso de que el bit de bandera sea 1 es igual a $2^{10} = 1\,024$. Más tarde se verá que es posible usar una instrucción *shift* (corrimiento) para dejar 0 o 1 en `$t1` de acuerdo con el valor del bit de bandera.

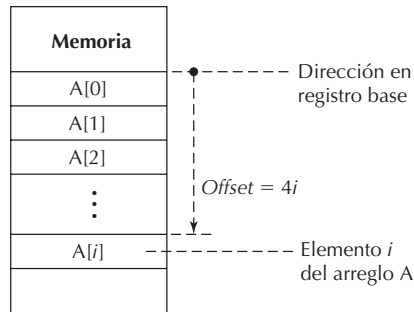
5.4 Instrucciones *load* y *store*

Las instrucciones *load* y *store* transfieren palabras completas (32 bits) entre memoria y registros. Cada una de esas instrucciones especifica un registro y una dirección de memoria. El registro, que es el destino de datos para carga y fuente de datos para almacenar, se especifica en el campo *rt* de una instrucción de formato I. La dirección de memoria se especifica en dos partes que se suman para producir la dirección: el entero con signo de 16 bits en la instrucción es un *offset* constante que se añade al valor base en el registro *rs*. En formato de lenguaje ensamblador, el registro *source/destination* (fuente/destino) *rt* se especifica primero, seguido por el *offset* constante y el registro base *rs* entre paréntesis. Al poner *rs* entre paréntesis, ello constituye un recuerdo de la notación de indexado $A(i)$ en lenguajes de alto nivel. He aquí dos ejemplos:

```
lw      $t0, 40($s3)      # carga mem[40+($s3)] en $t0
sw      $t0, A($s3)       # almacena ($t0) en mem[A+($s3)]
                        # "($s3)" significa "contenido de $s3"
```

El formato de instrucción de máquina para *lw* y *sw* se muestra en la figura 5.7 junto con la convención de direccionamiento de memoria. Note que el *offset* se puede especificar como un entero absoluto o a través de un nombre simbólico que se definió anteriormente. En el capítulo 7 se discutirá la definición de nombres simbólicos.

En la sección 6.4 se cubrirán las instrucciones *load* y *store* que se relacionan con tipos de datos distintos a las palabras. Aquí sólo se introduce otra instrucción *load* que permitirá la colocación de una constante arbitraria en un registro deseado. Una pequeña constante, que sea representable en 16 bits o menos, se puede cargar en un registro mediante una sola instrucción *addi*, cuyo otro operando es el registro *\$zero* (que siempre retiene 0). Una constante más grande se debe colocar en un registro en dos pasos; los 16 bits superiores se cargan en la mitad superior del registro a través de la instrucción



Nota acerca de la base y el *offset*:

La dirección de memoria es la suma de (*rs*) y un valor inmediato. Llamar a uno de ellos la base y al otro el *offset* es bastante arbitrario. Tendría mucho sentido interpretar la dirección $A(\$s3)$ como teniendo la base *A* y el *offset* (*\$s3*). Sin embargo, una base de 16 bits confina a una pequeña porción de espacio de memoria.

Figura 5.7 Instrucciones MiniMIPS *lw* y *sw* y sus convenciones de direccionamiento de memoria que permite el acceso simple a elementos de arreglos vía una dirección base y un *offset* ($offset = 4i$ lleva a la *i*-ésima palabra).

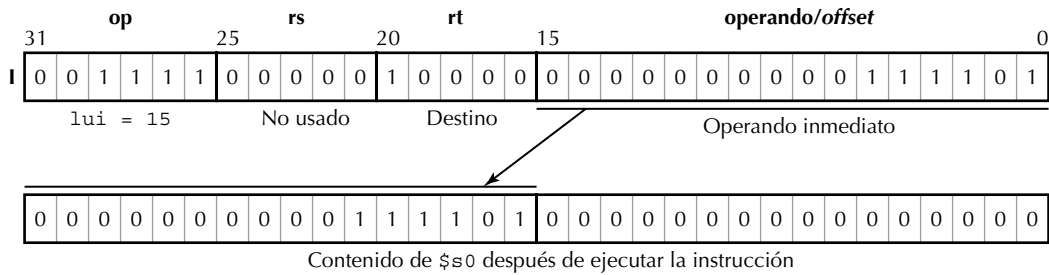


Figura 5.8 La instrucción `lui` permite cargar un valor arbitrario de 16 bits en la mitad superior de un registro mientras su mitad inferior se fija en 0.

load upper immediate (carga superior inmediata) `lui` y entonces los 16 bits inferiores se insertan mediante una *or inmediata* (`ori`). Esto es posible porque `lui` llena la mitad inferior del registro de destino con 0, de modo que cuando estos bits se operan OR con el operando inmediato, éste se copia en la mitad inferior del registro.

```
lui      $s0, 61          # El valor inmediato 61 (decimal)
                        # se carga en la mitad superior de $s0
                        # y los 16 bits inferiores se fijan en 0s
```

La figura 5.8 muestra la representación de máquina de la anterior instrucción `lui` y su efecto sobre el registro de destino `$s0`.

Observe que, aun cuando el par de instrucciones `lui` y `addi` a veces pueden lograr el mismo efecto que `lui` seguida por `ori`, esto último puede no ser el caso debido a la extensión de signo antes de añadir el operando inmediato de 16 bits al valor de registro de 32 bits.

Ejemplo 5.2: Carga de patrones arbitrarios de bits en registros Considere el patrón de bits que se muestra en la figura 5.6, correspondiente a una instrucción `addi`. Muestre cómo se puede cargar en el registro `$s0` este patrón de bits particular. ¿Y cómo se puede poner en `$s0` el patrón de bits que consiste todo de varios 1?

Solución: La mitad superior del patrón de bits de la figura 5.6 tiene la representación hexa 0x2110, mientras que la mitad inferior es 0x003d. Por ende, las siguientes dos instrucciones logran lo que se requiere:

```
lui      $s0, 0x2110      # pone mitad superior de patrón en $s0
ori      $s0, 0x003d      # pone mitad inferior de patrón en $s0
```

Estas dos instrucciones, con sus operandos inmediatos cambiados a 0xffff, podría colocar el patrón de todos los 1 en `$s0`. Una forma más simple y rápida sucede a través de la instrucción `nor`:

```
nor      $s0, $zero, $zero # porque (0V0)'=1
```

En representación de complemento a 2, el patrón de bits todos 1 corresponde al entero -1 . En consecuencia, luego de aprender, en el capítulo 9, que los MiniMIPS representan enteros en formato de complemento a 2, surgirán soluciones alternas para esta última parte.

5.5 Instrucciones *jump* y *branch*

En los MiniMIPS están disponibles las siguientes dos instrucciones *jump* (saltar) incondicionales:

```
j    verify    # ir a la localidad memoria llamada "verify"
jr   $ra       # ir a la localidad cuya dirección está en $ra;
                # $ra puede retener dirección de retorno de
                # un procedimiento (vea sección 6.1)
```

La primera instrucción representa un *jump* simple, que causa que la ejecución del programa proceda de la *localidad* cuya dirección numérica o simbólica se ofrece, en lugar de continuar con la siguiente instrucción en secuencia. *Jump register* (registro de salto), o *jr*, especifica un registro que retiene la dirección blanco de salto. Con frecuencia, este registro es *\$ra*, y se usa la instrucción *jr \$ra* para efectuar un retorno desde un procedimiento al punto desde donde se llamó a éste. Los procedimientos y su uso se discutirán en el capítulo 6. En la figura 5.9 se muestran las representaciones de máquina de las instrucciones *jump* de los MiniMIPS.

Para la instrucción *j*, se aumenta el campo *address* de 26 bits en la instrucción con 00 a la derecha y cuatro bits de orden superior del contador de programa (PC, por sus siglas en inglés) a la izquierda para formar una dirección completa de 32 bits. A esto se le llama *direccionamientoseudodirecto* (vea la sección 5.6). Dado que en la práctica, por lo general, el PC se incrementa muy pronto en el ciclo de ejecución de la instrucción (antes de la decodificación y ejecución de la instrucción), se supone que se usan 4 bits de orden superior de PC incrementado para aumentar el campo dirección de 26 bits.

Las instrucciones *branch* condicionales permiten transferir el control a una dirección específica cuando se satisface una condición de interés. En los MiniMIPS existen tres instrucciones *branch* básicas basadas en comparación para las que las condiciones constituyen un contenido de registro negativo y la igualdad o desigualdad de los contenidos de dos registros. En lugar de ofrecer múltiples instrucciones para otras comparaciones, los MiniMIPS ofrecen una instrucción de comparación de tipo R *set less than* (hacer menor que, *slt*) que establecen el contenido de un registro de destino específico a 1 si la relación “menor que” se sostiene entre los contenidos de dos registros dados, y se establece en 0 de motor modo. Para comparar con una constante, también se proporciona la versión inmediata de esta instrucción, a saber *slti*.

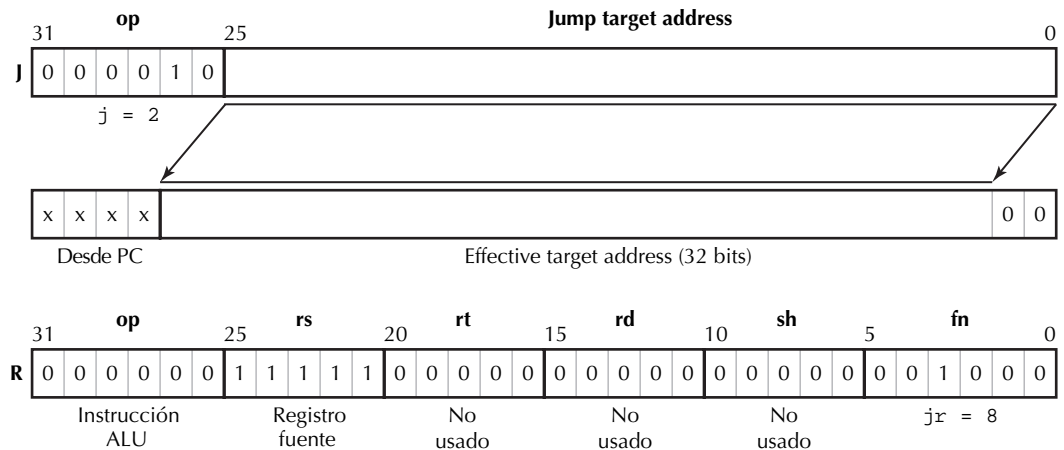


Figura 5.9 La instrucción *jump* *j* de los MiniMIPS es una instrucción de tipo J que se muestra junto con cómo se obtiene su dirección blanco efectiva. La instrucción *jump register* (*jr*) es de tipo R, siendo su registro efectivo usualmente *\$ra*.

```

bltz    $s1,L           # bifurcación en ($s1)< 0
beq     $s1,$s2,L       # bifurcación en ($s1)=$(s2)
bne     $s1,$s2,L       # bifurcación en ($s1)≠($s2)
slt     $s1,$s2,$s3     # si ($s2)<($s3), fija $s1 en 1
                        # sino fija $s1 en 0; esto es
                        # usualmente se sigue por beg o bne
slti    $s1,$s2,61      # si ($s2)<61, fija $s1 en 1
                        # sino fija $s1 en 0

```

La figura 5.10 muestra la representación de máquina de las tres instrucciones *branch* y las dos instrucciones comparación (*comparison*) apenas discutidas.

Para las tres instrucciones *bltz*, *beq* y *bne*, el direccionamiento relativo a PC se usa mediante el cual el *offset* con signo de 16 bits en la instrucción se multiplica por 4 y el resultado se suma al contenido de PC para obtener una dirección destino *branch* de 32 bits. Si la etiqueta especificada está muy lejos de alcanzarse mediante un *offset* de 16 bits (ocurrencia muy rara), el ensamblador sustituye *beq* *\$s0*, *\$s1*, *L1* con un par de instrucciones MiniMIPS:

```

        bne     $s1,$s2,L2      # pasa por alto jump si (s1)≠(s2)
        j       L1             # va a L1 si (s1)=(s2)
L2:     ...

```

Aquí, la notación “L2:” define la instrucción inespecífica que aparece en dicha línea, conteniendo la dirección simbólica L2. Por tanto, al escribir instrucciones *branch*, se pueden usar nombres simbólicos.

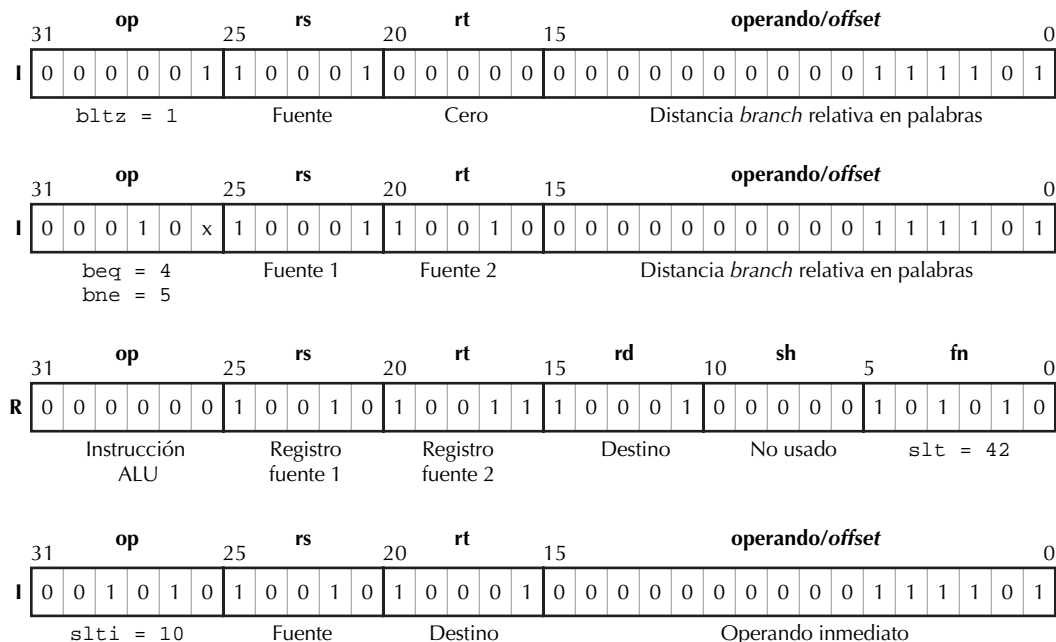


Figura 5.10 Instrucciones *branch* condicional (*bltz*, *beq*, *bne*) y *comparison* (*slt*, *slti*) de MiniMIPS.

licos libremente sin preocuparse si la dirección correspondiente se alcanza desde la instrucción actual mediante un *offset* de 16 bits.

Las instrucciones *branch* y *jump* se pueden usar para formar estructuras computacionales condicionales y repetitivas comparables a *if-then* (si-entonces) y *while* (mientras) de los lenguajes de alto nivel. El enunciado *if-then*

```
if (i == j) x = x + y;
```

se puede traducir al siguiente fragmento de programa en lenguaje ensamblador MiniMIPS:

```
        bne    $s1,$s2,endif    # bifurcación cuando i≠j
        add    $t1,$t1,$t2      # ejecuta la parte "then"
endif:  ...
```

Si la condición del enunciado *if* fuese $i < j$, entonces la primera instrucción en la secuencia anterior se sustituiría por las dos instrucciones siguientes (el resto no cambiaría):

```
slt     $t0,$s1,$s2            # fija $t0 en 1 si i<j
beq     $t0,$0,endif           # bifurcación si ($t0)=0; i.e., i≥j
                                # que es el inverso de i<j
```

Muchas otras bifurcaciones condicionales se pueden sintetizar de igual modo.

Ejemplo 5.3: Compilación de enunciados *if-then-else* Los lenguajes de alto nivel contienen una constructo *if-then-else* (si-entonces-sino) que permite la realización de dos cálculos, dependiendo de si se satisface una condición. Muestre una secuencia de instrucciones MiniMIPS que correspondan al siguiente enunciado *if-then-else*:

```
if (i <= j) x = x + 1; z = 1; else y = y - 1; z = 2 * z;
```

Solución: Esto es muy similar al enunciado *if-then*, excepto que se necesitan instrucciones correspondientes a la parte *else* y una forma de obviar la parte *else* después de que la parte *then* se ejecute.

```
        slt     $t0,$s2,$s1      # j<i? (inverso de condición)
        bne     $t0,$zero,else    # bifurcación en j<i a la parte "else"
        addi    $t1,$t1,1         # comienza "then" la parte: x=x+1
        addi    $t3,$zero,1       # z=1
        j       endif            # pasa por alto la parte "else"
else:    addi    $t2,$t2,-1        # comienza "else" la parte: y=y-1
        add     $t3,$t3,$t3       # z=z+z
endif:  ...
```

Observe que cada una de las dos secuencias de instrucción correspondientes a *then* y *else* del enunciado condicional pueden ser arbitrariamente largas.

El ciclo (*loop*) simple *while*

```
while (A[i] == k) i = i + 1;
```

se puede traducir al siguiente fragmento de programa en lenguaje ensamblador MiniMIPS, si supone que el índice *i*, la dirección de inicio de arreglo *A* y la constante de comparación *k* se encuentran en los registros *\$s1*, *\$s2* y *\$s3*, respectivamente:

```
loop: add    $t1,$s1,$s1    # calcula 2i en $t1
      add    $t1,$t1,$t1    # calcula 4i en $t1
      add    $t1,$t1,$s2    # coloca la dirección de A[i] en $t1
      lw     $t0,0($t1)     # carga el valor de A[i] en $t0
      bne    $t0,$s3,endwhl # sale del ciclo si A[i] ≠ k
      addi   $si,$si,1      # i = i + 1
      j      loop          # permanece en el ciclo
endwhl: ...
```

Observe que poner a prueba la condición *while-loop* requiere cinco instrucciones: dos para calcular el *offset* 4*i*, una para agregar el *offset* a la dirección base en *\$s2*, una para *fetch* el valor de *A[i]* de la memoria y una para verificar la igualdad.

Ejemplo 5.4: Ciclo con índice de arreglo y *goto* explícito Además de los ciclos *while*, los programas en lenguaje de alto nivel pueden contener ciclos en los que un enunciado *goto* (ir a) (condicional o incondicional) inicia la siguiente iteración. Muestre una secuencia de instrucciones MiniMIPS que realicen la función del siguiente ciclo de este tipo:

```
loop: i = i + step;
      sum = sum + A[i];
      if (i ≠ n) goto loop;
```

Solución: Esto es bastante similar a un ciclo *while* y, de hecho, se podría escribir como uno. Tomado del ejemplo *while-loop*, se escribe la siguiente secuencia de instrucciones, si supone que el índice *i*, la dirección de inicio del arreglo *A*, la constante de comparación *n*, *step*, y *sum* se encuentran en los registros *\$s1*, *\$s2*, *\$s3*, *\$s4* y *\$s5*, respectivamente:

```
loop: add    $s1,$s1,$s4    # i = i + step
      add    $t1,$s1,$s1    # calcula 2i en $t1
      add    $t1,$t1,$t1    # calcula 4i en $t1
      add    $t1,$t1,$s2    # coloca la dirección de A[i] en $t1
      lw     $t0,0($t1)     # carga el valor de A[i] en $t0
      add    $s5,$s5,$t0    # sum = sum + A[i]
      bne    $s1,$s3,loop    # si (i≠n) goto loop
```

Se supone que agregar repetidamente el incremento *step* al índice variable *i* eventualmente lo hará igual a *n*, de otro modo el ciclo nunca terminará.

5.6 Modos de direccionamiento

Modo de direccionamiento (addressing mode) es el método que especifica la ubicación de un operando dentro de una instrucción. Los MiniMIPS usan seis modos de direccionamiento, que se muestran en la figura 5.11 en forma esquemática y se describen de la forma siguiente.

1. *Direccionamiento implícito:* El operando viene de, o resulta de ir a, un lugar predefinido que no se especifica explícitamente en la instrucción. Un ejemplo de lo anterior se encuentra en la instrucción `jal` (que se introducirá en la sección 6.1), que siempre salva la dirección de la instrucción siguiente en la secuencia en el registro `$ra`.
2. *Direccionamiento inmediato:* El operando se proporciona en la instrucción en sí misma. Los ejemplos incluyen las instrucciones `addi`, `andi`, `ori` y `xori`, en las que el segundo operando (o, en realidad, su mitad inferior) se suministran como parte de la instrucción.
3. *Direccionamiento por registro:* El operando se toma de, o resulta colocado en, un registro específico. Las instrucciones tipo R en MiniMIPS especifican hasta tres registros como ubicaciones de sus operando(s) o resultado. Los registros se especifican mediante sus índices de cinco bits.
4. *Direccionamiento base:* El operando está en memoria y su ubicación se calcula al agregar un *offset* (entero con signo de 16 bits) al contenido de un registro base específico. Éste es el modo de direccionamiento de las instrucciones `lw` y `sw`.
5. *Direccionamiento relativo al PC:* Igual que el direccionamiento base, pero el registro siempre es el contador del programa y el *offset* se adiciona con dos 0 en el extremo derecho (siempre

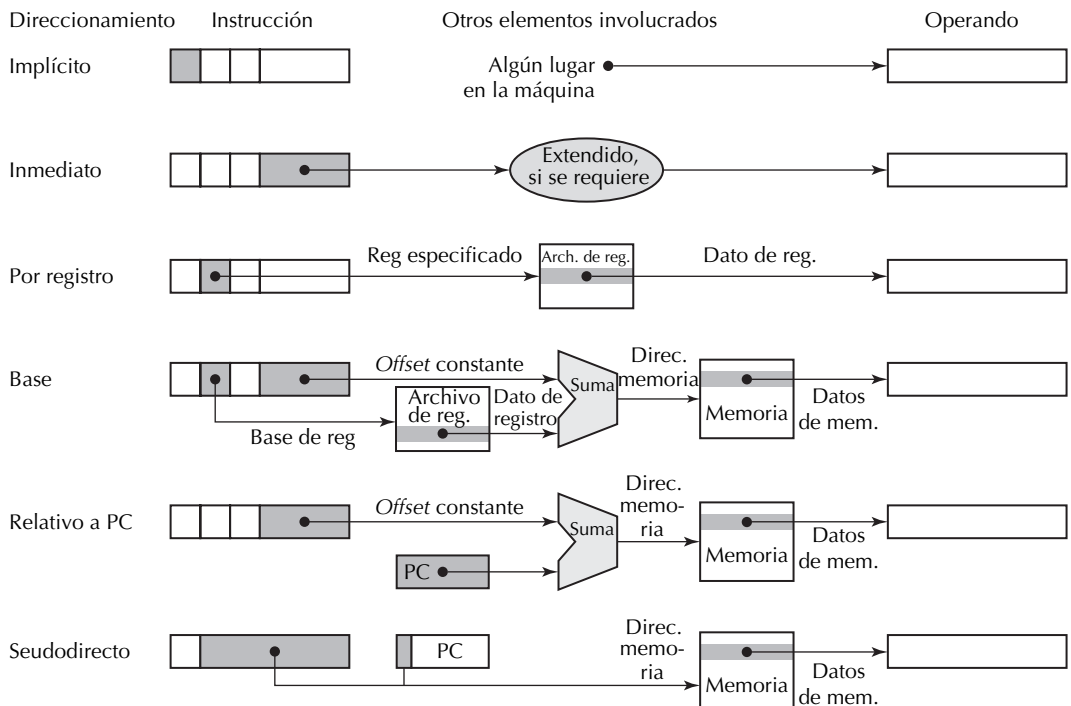


Figura 5.11 Representación esquemática de los modos de direccionamiento en MiniMIPS.

se usa el direccionamiento por palabra). Este es el modo de direccionamiento que se usa en las instrucciones *branch* y permite la bifurcación hacia otras instrucciones dentro de $\pm 2^{15}$ palabras de la instrucción actual.

6. *Direccionamientoseudodirecto*: En el direccionamiento directo, la dirección del operando se proporciona como parte de la instrucción. Para MiniMIPS esto último es imposible, dado que las instrucciones de 32 bits no tienen suficiente espacio para llevar direccionamientos completos de 32 bits. La instrucción *j* queda cerca del direccionamiento directo porque contiene 26 bits de la dirección destino de salto que se rellena con 00 en el extremo derecho y cuatro bits del contador de programa a la izquierda para formar una dirección completa de 32 bits; de ahí su denominación “seudodirecto”.

Los MiniMIPS tienen una arquitectura carga/almacenar: los operandos deben estar en los registros antes de que se les pueda procesar. Las únicas instrucciones MiniMIPS que se refieren a direcciones de memoria son *load*, *store* y *jump/branch*. Este esquema, en combinación con el limitado conjunto de modos de direccionamiento, permite que el hardware eficiente ejecute instrucciones MiniMIPS. Existen otros modos de direccionamiento. Algunos de estos modos alternos se discutirán en la sección 8.2. Sin embargo, los modos de direccionamiento que se usan en los MiniMIPS son más que adecuados por conveniencia en la programación y eficiencia en los programas resultantes.

En total, en este capítulo se introdujeron 20 instrucciones MiniMIPS. La tabla 5.1 resume estas instrucciones para su revisión y fácil referencia. Este conjunto de instrucciones es muy adecuado para componer programas bastante complejos. Con el propósito de que estos programas resulten modulares y eficientes, se necesitan los mecanismos adicionales que se tratan en el capítulo 6. Para hacer los programas ensamblador completos, se requieren las directivas ensamblador que se verán en el capítulo 7.

■ **TABLA 5.1** Las 20 instrucciones MiniMIPS vistas en el capítulo 5.

Clase	Instrucción	Uso	Significado	op	fn
Copia	Load upper immediate	lui <i>rt, imm</i>	$rt \leftarrow (imm, 0x0000)$	15	
	Add	add <i>rd, rs, rt</i>	$rd \leftarrow (rs) + (rt)$	0	32
Aritmética	Subtract	sub <i>rd, rs, rt</i>	$rd \leftarrow (rs) - (rt)$	0	34
	Set less than	slt <i>rd, rs, rt</i>	$rd \leftarrow \text{if } (rs) < (rt) \text{ then } 1 \text{ else } 0$	0	42
	Add immediate	addi <i>rt, rs, imm</i>	$rt \leftarrow (rs) + imm$	8	
	Set less than immediate	slti <i>rd, rs, imm</i>	$rd \leftarrow \text{if } (rs) < imm \text{ then } 1 \text{ else } 0$	10	
	AND	and <i>rd, rs, rt</i>	$rd \leftarrow (rs) \wedge (rt)$	0	36
Lógica	OR	or <i>rd, rs, rt</i>	$rd \leftarrow (rs) \vee (rt)$	0	37
	XOR	xor <i>rd, rs, rt</i>	$rd \leftarrow (rs) \oplus (rt)$	0	38
	NOR	nor <i>rd, rs, rt</i>	$rd \leftarrow ((rs) \vee (rt))'$	0	39
	AND immediate	andi <i>rt, rs, imm</i>	$rt \leftarrow (rs) \wedge imm$	12	
	OR immediate	ori <i>rt, rs, imm</i>	$rt \leftarrow (rs) \vee imm$	13	
	XOR immediate	xori <i>rt, rs, imm</i>	$rt \leftarrow (rs) \oplus imm$	14	
Acceso de memoria	Load word	lw <i>rt, imm(rs)</i>	$rt \leftarrow \text{mem}[(rs) + imm]$	35	
	Store word	sw <i>rt, imm(rs)</i>	$\text{mem}[(rs) + imm] \leftarrow (rt)$	43	
	Jump	j <i>L</i>	goto <i>L</i>	2	
Transferencia de control	Jump register	jrr <i>rs</i>	goto (<i>rs</i>)	0	8
	Branch less than 0	bltz <i>rs, L</i>	if (<i>rs</i>) < 0 then goto <i>L</i>	1	
	Branch equal	beq <i>rs, rt, L</i>	if (<i>rs</i>) = (<i>rt</i>) then goto <i>L</i>	4	
	Branch not equal	bne <i>rs, rt, L</i>	if (<i>rs</i>) \neq (<i>rt</i>) then goto <i>L</i>	5	

Ejemplo 5.5: Cómo encontrar el valor máximo en una lista de números Una lista de enteros que se almacena en memoria comienza en la dirección dada en el registro `$s1`. La longitud de la lista se proporciona en el registro `$s2`. Escriba una secuencia de instrucciones MiniMIPS de la tabla 5.1 para encontrar el entero más grande en la lista y copiarlo en el registro `$t0`.

Solución: Se examinan todos los elementos de la lista `A` y se usa `$t0` para mantener el entero más grande identificado hasta el momento (inicialmente, `A[0]`). En cada paso se compara un nuevo elemento de la lista con el valor en `$t0` y se actualiza el último si es necesario.

```

        lw      $t0,0($s1)      # inicializa máximo en A[0]
        addi    $t1,$zero,0     # inicializa índice i a 0
loop:   add     $t1,$t1,1        # incrementa el índice i en 1
        beq     $t1,$s2,done    # si todos los elementos examinados, salir
        add     $t2,$t1,$t1     # calcular 2i en $t2
        add     $t2,$t2,$t2     # calcular 4i en $t2
        add     $t2,$t2,$s1     # de la dirección de A[i] en $t2
        lw      $t3,0($t2)      # cargar valor de A[i] en $t3
        slt     $t4,$t0,$t3     # máximo < A[i]?
        beq     $t4,$zero,loop  # si no, repetir sin cambio
        addi    $t0,$t3,0       # si sí, A[i] es el nuevo máximo
        j       loop           # cambio completo; ahora repetir
done:   ...                    # continuación del programa

```

Observe que la lista se supuso no vacía y que contiene al menos el elemento `A[0]`.

PROBLEMAS

5.1 Formatos de instrucción

En los formatos de instrucción MIPS (figura 5.4), el campo *opcode* se puede reducir de seis a cinco bits y el campo *function* de seis a cuatro bits. Dado el número de instrucciones MIPS y diferentes funciones necesarias, estos cambios no limitarán el diseño; esto es: todavía se tendrán suficientes códigos de operación y códigos de función. Por tanto, los tres bits ganados se pueden usar para extender los campos *rs*, *rt* y *rd* de cinco a seis bits cada uno.

- Mencione dos efectos positivos de estos cambios. Justifique sus respuestas.
- Mencione dos efectos negativos de estos cambios. Justifique sus respuestas.

5.2 Otras instrucciones lógicas

Para ciertas aplicaciones, pueden ser útiles algunas otras operaciones lógicas de bits. Muestre cómo se pueden sintetizar las siguientes operaciones lógicas mediante las instrucciones MIPS tratadas en este capítulo. Intente usar tan pocas instrucciones como sea posible.

- NOT
- NAND
- XNOR
- NOR inmediata

5.3 Desbordamiento en suma

La suma de dos enteros de 32 bits puede no ser representable en 32 bits. En este caso, se dice que ocurrió un

desbordamiento (*overflow*). Hasta el momento, no se ha discutido cómo se detecta un desbordamiento ni cómo lidiar con él. Escriba una secuencia de instrucciones MiniMIPS que sumen dos números almacenados en los registros $\$s1$ y $\$s2$, almacena la suma (módulo 2^{32}) en el registro $\$s3$ y fija el registro $\$t0$ en 1 si ocurre un desbordamiento y en 0 de otro modo. *Sugerencia:* El desbordamiento es posible sólo con operandos del mismo signo; para dos operandos no negativos (negativo), si la suma que se obtiene es menor (mayor) que otro operando, ocurrió desbordamiento.

5.4 Multiplicación por una potencia pequeña de 2

Escriba una secuencia de instrucciones MiniMIPS (use sólo las de la tabla 5.1) para multiplicar el entero x almacenado en el registro $\$s0$ por 2^n , donde n es un entero pequeño no negativo almacenado en $\$s1$. El resultado se debe colocar en $\$s2$. *Sugerencia:* Use duplicación repetida.

5.5 Compilación de una sentencia switch/case

Una sentencia switch/case permite bifurcación multivía con base en el valor de una variable entera. En el ejemplo siguiente, la variable *switch* s se puede suponer uno de los tres valores en $[0, 2]$ y para cada caso se supone una acción diferente.

```
switch (s) {
    case 0:  a = a + 1; break;
    case 1:  a = a - 1; break;
    case 2:  b = 2 * b; break;
}
```

Muestre cómo se puede compilar tal sentencia en instrucciones ensamblador MiniMIPS.

5.6 Cálculo de valor absoluto

Escriba una secuencia de instrucciones MiniMIPS para colocar en el registro $\$s0$ el valor absoluto de un parámetro que se almacena en el registro $\$t0$.

5.7 Intercambio sin valores intermedios

Escriba una secuencia de instrucciones MiniMIPS para intercambiar (*swap*) los contenidos de los registros $\$s0$ y $\$s1$ sin perturbar el contenido de algún otro registro. *Sugerencia:* $(x \oplus y) \oplus y = x$.

5.8 Instrucción set size

Suponga que en MiniMIPS todas las instrucciones se tienen que codificar sólo con el uso de los campos *opcode* y *function*; esto es: ninguna otra parte de la instrucción debe portar información acerca del tipo de instrucción, aun cuando no se use para otros propósitos. Sean n_R , n_I y n_J el número admisible de instrucciones de tipos R, I y J, respectivamente. Escriba una ecuación desde la que se puede obtener el máximo valor posible de n_R , n_I o n_J , cuando se proporcionan las otras dos cuentas.

5.9 Bifurcación condicional

Modifique la solución al ejemplo 5.3, de modo que la condición puesta a prueba sea:

- a) $i < j$
- b) $i \geq j$
- c) $i + j \leq 0$
- d) $i + j > m + n$

5.10 Bifurcación condicional

Modifique la solución al ejemplo 5.4 de modo que la condición que se pone a prueba al final del ciclo sea:

- a) $i < n$
- b) $i \leq n$
- c) $\text{sum} \geq 0$
- d) $A[i] == 0$

5.11 Fragmento de programa misterioso

El siguiente fragmento de programa calcula un resultado $f(n)$ en el registro $\$s1$ cuando se proporciona el entero no negativo n en el registro $\$s2$. Agregue comentarios adecuados a las instrucciones y caracterice $f(n)$.

```
add    $t2, $s0, $s0
addi   $t2, $t2, 1
addi   $t0, $zero, 0
addi   $t1, $zero, 1
loop:  add    $t0, $t0, $t1
      beq    $t1, $t2, done
      addi   $t1, $t1, 2
      j      loop
done:  add    $s1, $zero, $t0
```

5.12 Otras bifurcaciones condicionales

Sólo con el uso de las instrucciones de la tabla 5.1, muestre cómo se puede obtener un efecto equivalente a las siguientes bifurcaciones condicionales:

- a) bgtz (bifurcación en mayor que 0)
- b) beqz (bifurcación en igual a 0)
- c) bnez (bifurcación en no igual a 0)
- d) blez (bifurcación en menor que o igual a 0)
- e) bgez (bifurcación en mayor que o igual a 0)

5.13 Identificación de valores extremos

Modifique la solución al ejemplo 5.5 de modo que conduzca a la identificación de:

- a) El entero más pequeño en la lista.
- b) El elemento de lista con el valor absoluto más grande.
- c) El elemento de lista cuyo byte menos significativo es el más grande.

5.14 Suma de un conjunto de valores almacenados en registros

Escriba la secuencia más corta posible de instrucciones MiniMIPS de la tabla 5.1 para sumar los contenidos de los ocho registros $\$s0$ – $\$s7$, y almacenar el resultado en $\$t0$. No se deben modificar los contenidos originales de los ocho registros.

- a) Suponga que los registros del $\$t0$ al $\$t9$ se pueden usar libremente para retener valores temporales.
- b) Suponga que ningún otro registro distinto a $\$t0$ se modifica en el proceso.

5.15 Reducción modular

Los enteros no negativos x y y se almacenan en los registros $\$s0$ y $\$s1$, respectivamente.

- a) Escriba una secuencia de instrucciones MiniMIPS que calculen $x \bmod y$, y almacene el resultado en $\$t0$. *Sugerencia:* Use sustracciones repetidas.
- b) Derive el número de instrucciones que se ejecutan en el cálculo de la parte a) en los casos mejor y peor.
- c) ¿Su respuesta a la parte a) conduce a un resultado razonable en el caso especial de $y = 0$?
- d) Aumente la secuencia de instrucciones en la parte a) de modo que también produzca $\lfloor x/y \rfloor$ en $\$t1$.
- e) Derive el número de instrucciones que se ejecutan en el cálculo de la parte d) en los casos mejor y peor.

5.16 Buffer cíclico

Un **buffer** cíclico (*ring buffer*) de 64 entradas se almacena en las localidades de memoria de la 5000 a la 5252. Las localidades de almacenamiento se llaman de

$L[0]$ a $L[63]$, y $L[0]$ se ve a continuación de $L[63]$ en forma circular. El registro $\$s0$ retiene la dirección de la primera entrada $B[0]$ del buffer. El registro $\$s1$ apunta a la entrada justo debajo de la última (es decir, donde se debe colocar el siguiente elemento $B[l]$, si se supone que la longitud del buffer actual es l). El llenado del buffer se indica mediante una bandera (*flag*) en $\$s2$ que retiene 1 si $l = 64$ y 0 de otro modo.

- a) Escriba una secuencia de instrucciones MiniMIPS para copiar el contenido de $\$t0$ en el buffer si no está lleno y ajuste la bandera de llenado para el caso de que se llene de ahí en adelante.
- b) Escriba una secuencia de instrucciones MiniMIPS para copiar el primer elemento de buffer en $\$t1$, siempre que este último no esté vacío, y ajuste $\$s0$ en concordancia.
- c) Escriba una secuencia de instrucciones MiniMIPS para buscar el buffer y ver si contiene un elemento x , cuyo valor, siempre que esté en $\$t2$, fije $\$t3$ en 1 si tal elemento se encuentra y en 0 si no es así.

5.17 Optimización de secuencias de instrucción

Considere la siguiente secuencia de tres sentencias de lenguaje de alto nivel: $X = Y + Z$; $Y = X + Z$; $W = X - Y$. Escriba una secuencia equivalente de instrucciones MiniMIPS y suponga que W se debe formar en $\$t0$, y que X , Y y Z están en las localidades de memoria cuyas direcciones se proporcionan en los registros $\$s0$, $\$s1$ y $\$s2$, respectivamente. Además, suponga lo siguiente:

- a) El cálculo se debe realizar exactamente como se especifica y cada enunciado se compila independientemente de los otros.
- b) Cualquier secuencia de instrucciones es aceptable en tanto el resultado final sea el mismo.
- c) Igual que la parte b), pero el compilador tiene acceso a registros temporales que se reservan para su uso.

5.18 Inicialización de un registro

- a) Identifique todas las posibles instrucciones MiniMIPS individuales de la tabla 5.1 que se pueden usar para inicializar un registro deseado a 0 (patrón de bits todos 0).
- b) Repita la parte a) para el patrón de bits todos 1.
- c) Caracterice el conjunto de todos los patrones de bits que se puedan colocar en un registro deseado con el uso de una sola instrucción de la tabla 5.1.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Kane92] Kane, G. y J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, 1992.
- [MIPS] Sitio Web de tecnologías de MIPS. Siga las ligas de arquitectura y documentación en: <http://www.mips.com/>
- [Patt98] Patterson, D. A. y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2a. ed., 1998.
- [Sail96] Sailer, P. M. y D. R. Kaeli, *The DLX Instruction Set Architecture Handbook*, Morgan Kaufmann, 1996.
- [SPIM] SPIM, simulador de software de descarga gratuita para un subconjunto del conjunto de instrucciones MIPS; más información en la sección 7.6.
<http://www.cs.wisc.edu/~larus/spim.html>
- [Swee99] Sweetman, D., *See MIPS Run*, Morgan Kaufmann, 1999.

PROCEDIMIENTOS Y DATOS

“Los matemáticos son como los franceses, cuando les hablas lo traducen a su lenguaje, y enseguida se transforma en algo completamente diferente.”

Johann Wolfgang von Goethe, 1829

“Existen demasiadas máquinas que dan los datos muy rápido.”

Robert Ludlum, Apocalypse Watch, 1996

TEMAS DEL CAPÍTULO

- 6.1 Llamadas de procedimiento simples
- 6.2 Uso de la pila para almacenamiento de datos
- 6.3 Parámetros y resultados
- 6.4 Tipos de datos
- 6.5 Arreglos y apuntadores
- 6.6 Instrucciones adicionales

En este capítulo continúa el estudio del conjunto de instrucciones MiniMIPS al examinar las instrucciones necesarias para llamadas de procedimiento y los mecanismos que se usan para pasar datos entre las rutinas llamador (*caller*) y llamada (*called*). En el proceso, aprenderá otros detalles de la arquitectura del conjunto de instrucciones y se familiarizará con algunas ideas importantes acerca de los tipos de datos, llamadas de procedimiento anidadas, utilidad de una pila, acceso a elementos de arreglo y aplicaciones de los apuntadores (*pointers*). Hacia el final del capítulo, conocerá suficientes instrucciones para escribir programas no triviales y útiles.

■ 6.1 Llamadas de procedimiento simples

Un *procedimiento* representa un subprograma que, cuando se *llama* (*inicia, invoca*) realiza una tarea específica, que quizás lleve a uno o más resultados, con base en los *parámetros de entrada* (*argumentos*) que se le proporciona y retornos al punto de llamada. En lenguaje ensamblador, un procedimiento se asocia con un nombre simbólico que denota su dirección de inicio. La instrucción `jal` en MiniMIPS utiliza específicamente para llamadas de procedimiento: realiza la transferencia de control (*jump* incondicional) a la dirección de inicio del procedimiento, pero al mismo tiempo salva la dirección de retorno en el registro `$ra`.

```

jal    proc           # salta a la ubicación "proc" y liga
                        # "liga" significa "salvar la dirección
                        # de retorno" (PC)+4 en $ra ($31)

```

Observe que, mientras que una instrucción `jal` se ejecuta, el contador de programa retiene su dirección; por tanto, $(PC) + 4$ designa la dirección de la siguiente instrucción después de `jal`. El formato de instrucción de máquina para `jal` es idéntico al de la instrucción `jump` que se muestra en la parte superior de la figura 5.9, excepto que el campo *opcode* contiene 3.

Usar un procedimiento involucra la siguiente secuencia de acciones.

1. Poner argumentos en lugares conocidos al procedimiento (registros $\$a0$ – $\$a3$).
2. Transferir control al procedimiento y salvar la dirección de retorno (`jal`).
3. Adquirir espacio de almacenamiento, si se requiere, para uso del procedimiento.
4. Realizar la tarea deseada.
5. Poner los resultados en lugares conocidos al programa que llama (registros $\$v0$ – $\$v1$).
6. Regresar el control al punto de llamada (`jr`).

El último paso se realiza mediante la instrucción `jump register` (registro de salto) (figura 5.9):

```

jr     rs             # ir a ubic cuya dirección está en rs

```

Más adelante se verá como se pueden acomodar los procedimientos con más de cuatro palabras de argumentos y dos palabras de resultados.

Conforme el procedimiento se ejecuta, utiliza registros para retener operandos y resultados parciales. Después de regresar de un procedimiento, el programa que llama puede esperar razonablemente encontrar sus propios operandos y resultados parciales donde estuvieron antes de llamar al procedimiento. Si este requisito se forzase para todos los registros, cada procedimiento necesitaría salvar el contenido original de cualquier registro que use y luego restaurarlo antes de su terminación. La figura 6.1 muestra este salvado y restauración dentro del procedimiento llamado, así como la preparación para llamar al procedimiento y continuar en el programa principal.

Con el propósito de evitar el uso del sistema asociado con un gran número de operaciones de salvado y restauración de registros durante las llamadas de procedimiento, se usa la siguiente convención. Un procedimiento puede usar libremente los registros $\$v0$ – $\$v1$ y $\$t0$ – $\$t9$ sin tener que salvar sus contenidos originales; en otras palabras, un programa llamado no debe esperar que cualesquiera valores colocados en estos 12 registros permanezcan sin cambio después de llamar el procedimiento. Si el programa que llama tiene algún valor en estos registros, debe salvarlos a otros registros o a la memoria antes de llamar un procedimiento. Esta división de responsabilidad entre los programas que llaman y llamado es muy

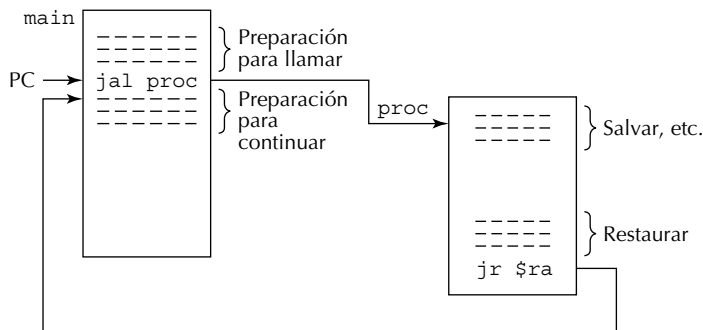


Figura 6.1 Relación entre el programa principal y un procedimiento.

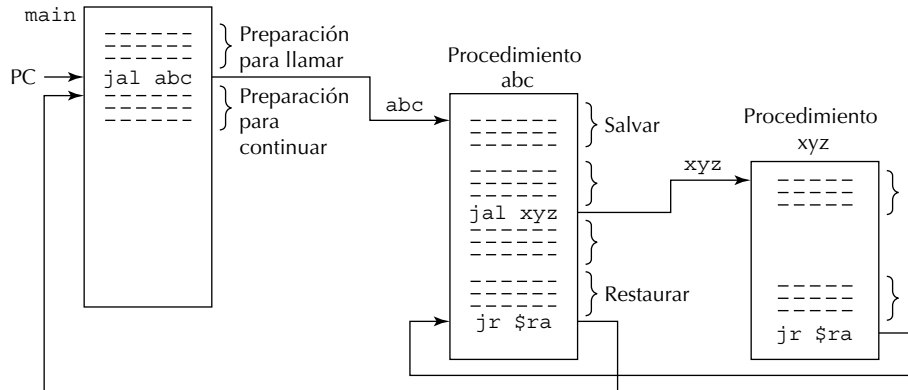


Figura 6.2 Ejemplo de llamadas de procedimiento anidadas.

sensible. Ella permite, por ejemplo, llamar un procedimiento simple sin salvar y restaurar en absoluto. Como se indica en la figura 5.2, el programa que llama (*caller*) puede esperar que los valores en los siguientes registros no se perturben al llamar un procedimiento: $\$a0$ – $\$a3$, $\$0$ – $\$a7$, $\$gp$, $\$sp$, $\$ft$, $\$ra$. Un procedimiento (programa *llamado*, *callee*) que modifica alguno de éstos los debe restaurar a sus valores originales antes de terminar. Para resumir, la convención de salvado de registro en MiniMIPS es como sigue:

Registros salvados por *caller*: $\$v0$ – $\$v1$, $\$t0$ – $\$t9$

Registros salvados por *callee*: $\$a0$ – $\$a3$, $\$s0$ – $\$s7$, $\$gp$, $\$sp$, $\$fp$, $\$ra$

En la sección 6.2 se discutirán los usos de los registros $\$gp$, $\$sp$ y $\$fp$. En todos los casos, salvar el contenido de un registro sólo se requiere si dicho contenido lo usará en lo futuro el *caller/callee*. Por ejemplo, salvar $\$ra$ sólo se requiere si el *callee* llamará otro procedimiento que entonces sobrescribiría la actual dirección de retorno con su propia dirección de retorno. En la figura 6.2, el procedimiento *abc*, a su vez, llama al procedimiento *xyz*. Note que, antes de llamar a *xyz*, el procedimiento que llama *abc* realiza algunas acciones preparatorias, que incluyen poner los argumentos en los registros $\$a0$ – $\$a3$ y salvar cualquiera de los registros $\$v0$ – $\$v1$ y $\$t0$ – $\$t9$ que contiene datos útiles. Después de regresar de *xyz*, el procedimiento *abc* puede transferir cualquier resultado en $\$v0$ – $\$v1$ a otros registros. Esto último se necesita, por ejemplo, antes de que se llame otro procedimiento, para evitar sobrescribir los resultados del procedimiento previo por el siguiente procedimiento. Note que, con cuidado adecuado, también se pueden usar los registros $\$v0$ y $\$v1$ para pasar parámetros a un procedimiento sin usar la pila.

El diseño de un procedimiento se ilustra mediante dos ejemplos simples. Más adelante, en este capítulo, se darán procedimientos más interesantes y realistas.

Ejemplo 6.1: Procedimiento para encontrar el valor absoluto de un entero Escriba un procedimiento MiniMIPS que acepte un parámetro entero en el registro $\$a0$ y regresa su valor absoluto en $\$v0$.

Solución: El valor absoluto de x es $-x$ si $x < 0$ y x de otro modo.

```

abs:  sub  $v0,$zero,$a0    # fijar -($a0) en $v0; en caso ($a0)<0
      bltz $a0,done        # si ($a0)<0 entonces
      add  $v0,$a0,$zero    # sino fijar ($a0) en $v0
done:  jr   $ra             # regresar a programa llamante
  
```

En la práctica, rara vez se usan tales procedimientos cortos debido al excesivo uso de máquina que representan. En este ejemplo se tienen entre tres y cuatro instrucciones de uso de máquina para tres instrucciones de cálculo útil. Las instrucciones adicionales son `jal`, `jr`, una instrucción para colocar el parámetro en `$a0` antes de la llamada y quizás una instrucción para mover el resultado regresado (se dice “quizás” porque el resultado podía usarse directamente fuera del registro `$v0`, si se hace antes de la siguiente llamada de procedimiento).

Ejemplo 6.2: Procedimiento para encontrar el más grande de tres enteros Escriba un procedimiento MiniMIPS que acepte tres parámetros enteros en los registros `$a0`, `$a1` y `$a2` y regresa el máximo de los tres valores en `$v0`.

Solución: Comience por suponer que `$a0` retiene el valor más grande, compare este valor contra cada uno de los otros dos valores y sustitúyalo cuando se encuentre un valor más grande.

```
max:  add    $v0,$a0,$zero    # copia ($a0) en $v0; más grande hasta el
                                momento
        sub    $t0,$a1,$v0    # calcula ($a1) - ($v0)
        bltz   $t0,okay       # si ($a1) - ($v0) < 0 entonces no cambia
        add    $v0,$a1,$zero  # de otro modo ($a1) es más grande hasta
                                el momento
okay:  sub    $t0,$a2,$v0    # calcula ($a2) - ($v0)
        bltz   $t0,done       # si ($a2) - ($v0) < 0 entonces no cambia
        add    $v0,$a2,$zero  # de otro modo ($a2) es más grande que
                                todos
done:  jr     $ra             # regresa al programa llamante
```

También aquí se aplican los comentarios al final de la solución del ejemplo 6.1. Vea si puede derivar el *overhead* en este caso.

■ 6.2 Uso de la pila para almacenamiento de datos

Los mecanismos y convenciones discutidos en la sección 6.1 son adecuados para procedimientos que aceptan hasta cuatro argumentos, regresan hasta dos resultados y usan una docena más o menos de valores intermedios en el curso de sus cálculos. Más allá de estos límites, o cuando el procedimiento debe él mismo llamar otro procedimiento (por lo que necesita el salvado de algunos valores), se necesita espacio de almacenamiento adicional. Un mecanismo común para salvar cosas o hacer espacio para datos temporales que requiere un procedimiento es el uso de una estructura de datos dinámicos conocida como *stack* (pila).

Antes de discutir la *stack* y cómo resuelve el problema de almacenamiento de datos para los procedimientos, observe las convenciones para usar el espacio de dirección de memoria en MiniMIPS. La figura 6.3 muestra un mapa de la memoria MiniMIPS y el uso de los tres registros apuntadores `$gp`, `$sp` y `$fp`. La segunda mitad de la memoria MiniMIPS (comenzando con la dirección hexa `0x80000000`) se usa para I/O mapeado por memoria y, por tanto, no está disponible para almacenar programas o datos. En el capítulo 22 se discutirá el I/O (direccionamiento entrada/salida) mapeado por memoria. La primera mitad de esta última, que se extiende desde la dirección 0 hasta la dirección `0x7fffffff`, se divide en cuatro segmentos del modo siguiente:

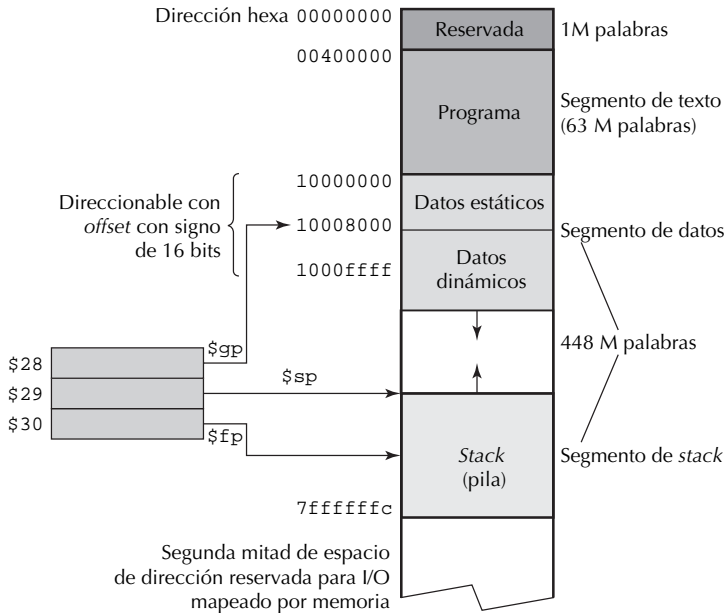


Figura 6.3 Panorama global del espacio de dirección de memoria en MiniMIPS.

Las primeras 1 M palabras (4 MB) se reservan para uso del sistema.

Las siguientes 63 M palabras (252 MB) retienen el texto del programa a ejecutar.

Comenzando en la dirección hexa 0×10000000 , se almacenan los datos del programa.

Comenzando en la dirección hexa $0 \times 7ffffffc$, y con crecimiento hacia atrás, está la *stack*.

Los datos dinámicos del programa y la *pila* pueden crecer en tamaño hasta la memoria máxima disponible. Si se fija el apuntador de registro global (\$gp) para retener la dirección 0×10008000 , entonces los primeros 2^{16} bytes de los datos del programa se vuelven fácilmente accesibles a través de direccionamiento base de la forma *imm* (\$gp), donde *imm* es un entero con signo de 16 bits.

Lo anterior conduce al uso del segmento de la *pila* y su registro del apuntador de lo alto de la pila \$sp (o *apuntador de pila*, para abreviar). La *pila* es una estructura de datos dinámicos en la que los datos se pueden colocar y recupera en orden *last-in, first-out* (último en entrar, primero en salir). Se puede ligar a la pila de charolas en una cafetería. Conforme las charolas se limpian y están listas para usarse, se colocan en lo alto de la pila de charolas; de igual modo, los clientes toman sus charolas de lo alto de la pila. La última charola colocada en la pila es la primera que uno toma.

Los elementos de datos se añaden a la *pila* al empujarlas en la *pila* y se recuperan al sacarlos. Las operaciones *push* (empujar) y *pop* (sacar) de la *pila* se ilustran en la figura 6.4 para una *pila* que tiene elementos de datos *b* y *a* como sus dos elementos altos. El apuntador de pila apunta al elemento alto de la pila que actualmente retiene *b*. Esto último significa que la instrucción `lw $t0, 0($sp)` causa que el valor *b* se copie en \$t0 y `sw $t1, 0($sp)` hace que *b* se sobrescriba con el contenido de \$t1. Por ende, un nuevo elemento *c* actualmente en el registro \$t4 se puede empujar en la *pila* mediante las siguientes dos instrucciones MiniMIPS:

```
push:  addi  $sp, $sp, -4
       sw    $t4, 0($sp)
```

Note que es posible invertir el orden de las dos instrucciones si se usara la dirección $-4($sp)$ en lugar de $0($sp)$. Para sacar el elemento *b* de la *stack* en la figura 6.4, se usa una instrucción `lw` para

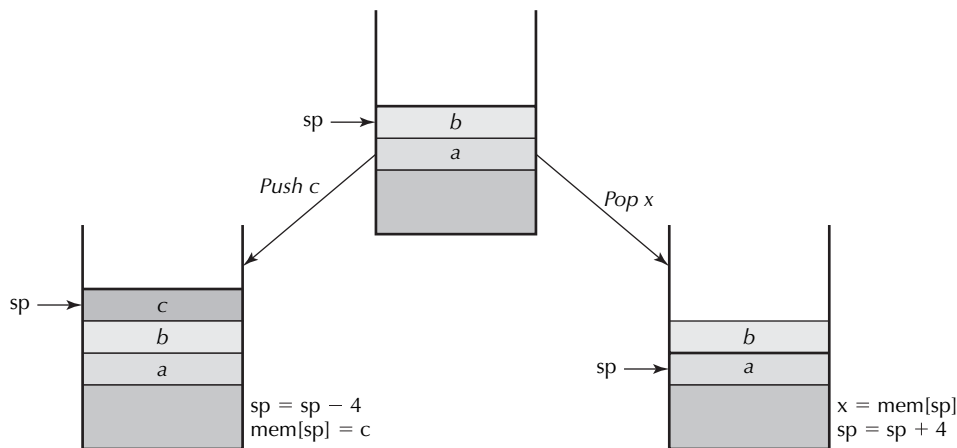


Figura 6.4 Efectos de las operaciones *push* y *pop* en una pila.

copiar *b* en un registro deseado y el apuntador de pila se incrementa en 4 para apuntar al siguiente elemento de la pila:

```
pop:  lw    $t5, 0($sp)
      addi   $sp, $sp, 4
```

De nuevo se podría ajustar el apuntador de *pila* antes de copiar su elemento alto. Observe que una operación *pop* no borra de la memoria al antiguo elemento alto *b*; *b* todavía está donde se encontraba antes de la operación *pop* y de hecho sería accesible a través de la dirección $-4(\$sp)$. Sin embargo, la localidad que retiene *b* ya no se considera más parte de la *pila*, cuyo elemento alto ahora es *a*.

Aun cuando técnicamente una *pila* es una estructura de datos *last-in, first-out*, de hecho se puede tener acceso a cualquier elemento dentro de la *pila* si se conoce su orden relativo desde arriba. Por ejemplo, se puede acceder al elemento justo abajo de lo alto de la *pila* (su segundo elemento) con el uso de la dirección de memoria $4(\$sp)$, y el quinceavo elemento de la *pila* está en la dirección $56(\$sp)$. De igual modo, no se necesita remover los elementos uno a la vez. Si ya no se necesitan los diez elementos altos de una *pila*, se les puede remover todos a la vez al incrementar el apuntador de *pila* en 40.

6.3 Parámetros y resultados

En esta sección se responden las siguientes preguntas no resueltas relacionadas con los procedimientos:

1. ¿Cómo se pueden pasar más de cuatro parámetros de entrada a un procedimiento o recibir más de dos resultados de él?
2. ¿Dónde salva un procedimiento sus propios parámetros y resultados intermedios cuando llama a otro procedimiento (llamadas anidadas)?

En ambos casos se usa la pila. Antes de una llamada de procedimiento, el programa llamante empuja los contenidos de cualesquier registros que necesite salvar en lo alto de la *stack* y sigue esto con cualesquiera argumentos adicionales para el procedimiento. El procedimiento también puede acceder a estos argumentos en la *stack*. Después de que el procedimiento termina, el programa llamante espera encontrar el apuntador de *pila* imperturbado; en consecuencia, le permite restaurar los registros salvados a

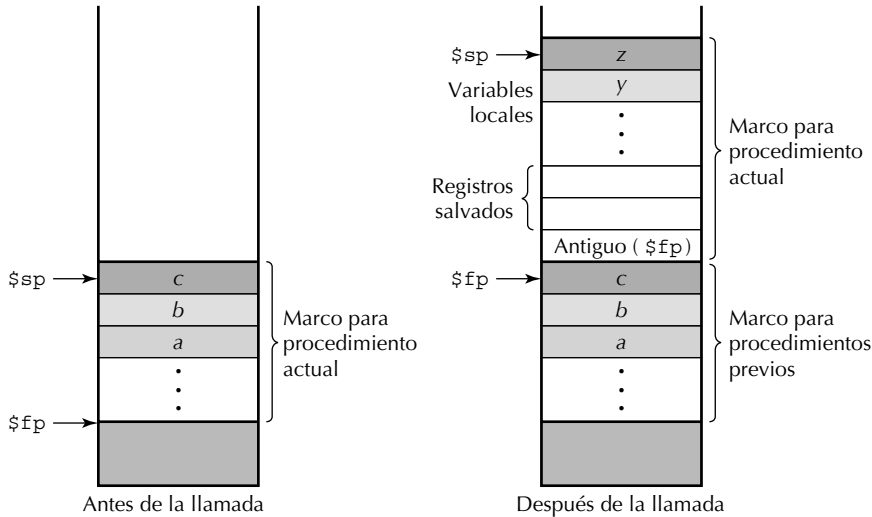


Figura 6.5 Uso de la *pila* para un procedimiento.

sus estados originales y procede con sus propios cálculos. En consecuencia, un procedimiento que usa la *pila* mediante la modificación del apuntador de *pila* debe salvar el contenido de éste en el principio y, al final, restaurar $\$sp$ a su estado original. Eso se hace mediante el copiado del apuntador de *pila* en el registro del apuntador de marco $\$fp$. Sin embargo, antes de hacer esto último, se deben salvar los contenidos antiguos del apuntador de marco. Por tanto, mientras se ejecuta un procedimiento, $\$fp$ debe retener el contenido de $\$fp$ justo antes de que se llame; $\$fp$ y $\$sp$ “marcan” juntos el área de la *pila* que está en uso por el procedimiento actual (figura 6.5).

En el ejemplo que se muestra en la figura 6.5, los tres parámetros a , b y c pasan al procedimiento al colocarlos en lo alto de la *pila* justo antes de llamar al procedimiento. El procedimiento primero empuja los contenidos de $\$fp$ en la *pila*, copia el apuntador de *pila* en $\$fp$, empuja los contenidos de los registros que necesitan salvarse en la *pila*, usa la *pila* para retener aquellas variables locales temporales que no se pueden retener en los registros, y así por el estilo. Más tarde puede llamar a otro procedimiento al colocar argumentos, como y y z , en lo alto de la *pila*. Cada procedimiento en una secuencia anidada deja imperturbada la parte de la *pila* que pertenece a los llamadores previos y usa la *pila* que comienza con la rendija vacía sobre la entrada a la que se apunta con $\$sp$. Hasta la terminación de un procedimiento, el proceso se invierte; las variables locales se sacan de la *pila*, los contenidos de registro se restauran, el apuntador del marco se copia en $\$sp$ y, finalmente, $\$fp$ se restaura a su estado original.

A lo largo de este proceso, $\$fp$ proporciona un punto de referencia estable para direccionar palabras de memoria en la porción de la *pila* correspondiente al procedimiento actual. También ofrece una forma conveniente de regresar $\$sp$ a su valor original en la terminación del procedimiento. Las palabras en el marco actual tienen direcciones $-4(\$fp)$, $-8(\$fp)$, $-12(\$fp)$, etc. Mientras que el apuntador de *pila* cambia en el curso de la ejecución del procedimiento, el apuntador de marco retiene una dirección fija a todo lo largo. Observe que el uso de $\$fp$ es opcional. Un procedimiento que no llama él mismo a otro procedimiento puede usar la *pila* por cualquier razón sin cambiar jamás el apuntador de *pila*. Simplemente almacena datos en, y accede, a los elementos de *pila* más allá del alto de *pila* actual con el uso de las direcciones de memoria $-4(\$sp)$, $-8(\$sp)$, $-12(\$sp)$, etc. De esta forma, el puntero de *pila* no necesita salvarse porque nunca se modifica por el procedimiento.

Por ejemplo, el siguiente procedimiento salva los contenidos de \$fp, \$ra y \$s0 en la *pila* al principio y los restaura justo antes de la terminación:

```
proc: sw      $fp,-4($sp)  # salva el antiguo apuntador de marco
      addi    $fp,$sp,0    # salva ($sp) en $fp
      addi    $sp,$sp,-12  # crea tres espacios en lo alto de la pila
      sw      $ra,4($sp)   # salva ($ra) en segundo elemento de pila
      sw      $s0,0($sp)   # salva ($s0) en el elemento alto de pila
      .
      .
      .
      lw      $s0,0($sp)   # pone el elemento alto de pila en $s0
      lw      $ra,4($sp)   # pone segundo elemento de pila en $ra
      addi    $sp,$fp,0    # restaura $sp a estado original
      lw      $fp,-4($sp)  # regresa del procedimiento $fp a estado
                           original
      jr      $ra          # regresa del procedimiento
```

Si se sabe que este procedimiento ni llama a otro procedimiento ni necesita la *pila* para propósito alguno distinto al de salvar los contenidos del registro \$s0, se podría lograr el mismo fin sin salvar \$fp o \$ra o incluso ajustar el apuntador de *pila*:

```
proc: sw      $s0,-4($sp)  # salva ($s0) sobre alto de pila
      .
      .
      .
      lw      $s0,-4($sp)  # pone el elemento alto de pila en $s0
      jr      $ra          # regresa del procedimiento
```

Lo anterior reduce sustancialmente el uso de máquina de la llamada de procedimiento.

■ **6.4 Tipos de datos**

En la sección 5.1, y en la figura 5.2, se hizo referencia a varios tamaños de datos (*byte*, *palabra*, *doblepalabra*) en MiniMIPS. Sin embargo, subsecuentemente sólo se consideraron instrucciones que trataban con operandos de tamaño de palabra. Por ejemplo, lw y sw transfieren palabras entre registros y localidades de memoria, las instrucciones lógicas operan sobre operandos de 32 bits que se visualizan como cadenas de bits, y las instrucciones aritméticas operan sobre enteros con signo con tamaño de palabra. Otros tamaños de datos usados de manera común son *halfword* (media palabra) (16 bits) y *quadword* (cuádruple palabra) (128 bits). Ninguno de estos dos tipos se reconoce en MiniMIPS. Aun cuando los operandos inmediatos en MiniMIPS son halfwords, son extendidos en signo (para el caso de enteros con signo) o extendidos cero (para el caso de cadenas de bits) a 32 bits antes de operar con ellos. La única excepción es la instrucción lui, que carga un operando inmediato de 16 bits directamente en la mitad superior de un registro.

Mientras que el *tamaño de datos* se refiere al número de bits en una pieza particular de datos, el *tipo de datos* refleja el significado asignado a un elemento de datos. Los MiniMIPS tienen los siguientes tipos de datos, y cada uno se proporciona con todos los posibles tamaños que están disponibles:

Entero con signo:	byte	palabra	
Entero sin signo	byte	palabra	
Número en punto flotante:		palabra	doble palabra
Cadena de bits:	byte	palabra	doble palabra

Observe que no hay entero doble palabra o número de punto flotante con tamaño de byte: lo primero sería bastante factible y se excluye para simplificar el conjunto de instrucciones, mientras que lo último no es factible. Los números con punto flotante se discutirán en el capítulo 9; hasta entonces no se dirá nada acerca de este tipo de datos. De este modo, en el resto de esta sección se discuten las diferencias entre los enteros signados y no signados y algunas nociones relacionadas con cadenas de bits.

En los MiniMIPS, los enteros con signo se representan en formato de complemento a 2. Los enteros pueden asumir valores en el intervalo $[-2^7, 2^7 - 1] = [-128, 127]$ en formato de ocho bits y $[-2^{31}, 2^{31} - 1] = [-2\,147\,483\,648, 2\,147\,483\,647]$ en formato de 32 bits. En representación de máquina, el signo de un entero con signo es evidente a partir de su bit más significativo: 0 para “+” y 1 para “-”. Esta representación se estudiará en la sección 9.4. Los enteros sin signo son sólo números binarios ordinarios con valores en $[0, 2^8 - 1] = [0, 255]$ en formato de ocho bits y $[0, 2^{32} - 1] = [0, 4\,294\,967\,295]$ en formato de 32 bits. Las reglas de aritmética con números con signo y sin signo son diferentes; por tanto, MiniMIPS ofrece un número de instrucciones para aritmética sin signo. Esto último se presentará en la sección 6.6. Cambiar el tamaño de datos también se realiza de modo diferente para números con signo y sin signo. Ir de un formato más estrecho a otro más ancho se realiza mediante extensión de signo (repetir el bit de signo en las posiciones adicionales) para números signados y mediante extensión cero para números no signados. Los siguientes ejemplos ilustran la diferencia:

Tipo	Número de ocho bits	Valor	Versión 32 bits del mismo número
Sin signo	0010 101	43	0000 0000 0000 0000 0000 0000 0010 1011
Sin signo	1010 101	17	0000 0000 0000 0000 0000 0000 1010 1011
Con signo	0010 1011	+43	0000 0000 0000 0000 0000 0000 0010 1011
Con signo	1010 1011	-85	1111 1111 1111 1111 1111 1111 1010 1011

En ocasiones, se quiere procesar cadenas de bytes (por ejemplo, pequeños enteros o símbolos de un alfabeto) en oposición a palabras. Dentro de poco se presentarán las instrucciones *load* y *store* para datos con tamaño de bytes. A partir del argumento anterior se deduce que un byte con signo y un entero de ocho bits sin signo se cargarán de manera diferente en un registro de 32 bits. Como consecuencia de lo anterior, la instrucción *load byte*, al igual que muchas otras instrucciones con propiedades similares, tendrá dos versiones, para valores con signo y sin signo.

Un uso importante para elementos de datos con tamaño de byte es representar los símbolos de un alfabeto que consiste de letras (mayúsculas y minúsculas), dígitos, signos de puntuación y otros símbolos necesarios. En notación hexa, un byte de ocho bits se convierte en un número de dos dígitos. La tabla 6.1 muestra cómo se asignan tales números hexa de dos dígitos (bytes de ocho bits) para representar los símbolos del American Standard Code for Information Interchange (ASCII: código estándar estadounidense para intercambio de información). En realidad, la tabla 6.1 define un código de siete bits y deja la mitad derecha de la tabla de código sin especificar. La mitad no especificada se puede usar para otros símbolos necesarios en varias aplicaciones. Aun cuando “ASCII 8 bits/7 bits” es el uso correcto, a veces se habla acerca del “código ASCII 8 bits/7 bits” (es decir, se repite “código” por claridad).

Cuando se involucra un alfabeto con más de 256 símbolos, cada símbolo requiere más de un byte para su representación. El siguiente tamaño de datos más largo, una media palabra, se puede usar para codificar un alfabeto con hasta $2^{16} = 65\,536$ símbolos. Esto es más que adecuado para casi todas las aplicaciones de interés. *Unicode*, que representa un estándar internacional, usa medias palabras de 16 bits para representar cada símbolo.

■ **TABLA 6.1** ASCII (American Standard Code for Information Interchange)^{1,2}

	0	1	2	3	4	5	6	7	8-9	a-f
0	NUL	DLE	SP ⁴	0	@	P	`	p		
1	SOH	DC1	!	1	A	Q	a	q		
2	STX	DC2	"	2	B	R	b	r	M	
3	ETX	DC3	#	3	C	S	c	s	Á	Á
4	EOT	DC4	\$ ³	4	D	T	d	t	S	S
5	ENQ	NAK	%	5	E	U	e	u		
6	ACK	SYN	&	6	F	V	f	v	C	S
7	BEL	ETB	'	7	G	W	g	w	O	Í
8	BS	CAN	(8	H	X	h	x	N	M
9	HT	EM)	9	I	Y	i	y	T	B
a	LF	SUB	*	:	J	Z	j	z	R	O
b	VT	ESC	+	;	K	[k	{	O	L
c	FF	FS	,	<	L	\	l		L	O
d	CR	GS	-	=	M]	m	}	E	S
e	SO	RS	.	>	N	^	n	~	S	
f	SI	US	/	?	O	_	o	DEL ⁴		

1: El código ASCII de ocho bits se forma como (column# row#)_{hex}; por ejemplo, “+” es (2b)_{hex}.

2: Las columnas 0–7 definen el código ASCII 7 bits; por ejemplo, “+” es (010 1011)_{dos}.

3: El código ISO difiere del ASCII sólo en el símbolo monetario.

4: Las columnas 0–1 y 8–9 contienen caracteres de control, que se mencionan en orden alfabético.

ACK	Reconocimiento	EM	Fin de medio	GS	Separador de grupo	SOH	Comienzo encabezado
BEL	Campana	ENQ	Consulta	HT	Tabulador horizontal	SP	Espacio
BS	Retroceder un espacio	EOT	Fin de trans.	LF	Alimentación de línea	STX	Comienzo de texto
CAN	Cancelar	ETB	Fin bloque trans.	NAK	Rec. negativo	SUB	Sustituto
CR	Retorno de carro	ETX	Fin de texto	NUL	Nulo	SYN	Libre síncrono
DCi	Control de dispositivo	FF	Alimentación de forma	RS	Separador de registro	US	Separador de unidad
DLE	Escape de liga de datos	FS	Separador de archivo	SI	Shift in	VT	Tabulador vertical
				SO	Shift out		

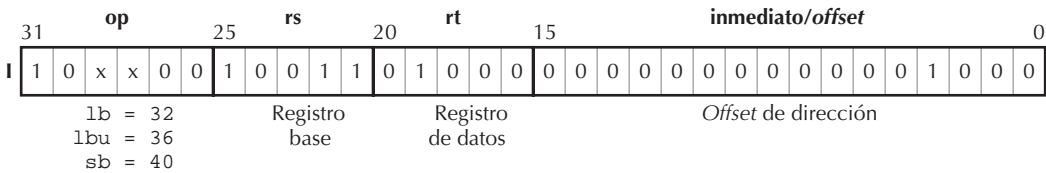


Figura 6.6 Instrucciones *load* y *store* para elementos de datos con tamaño de byte.

En MiniMIPS, las instrucciones *load byte*, *loady byte unsigned* y *store byte* permiten transferir bytes entre memoria y registros:

```
lb      $t0,8($s3)      # carga rt con byte mem[8+($s3)];
                        # extensión de signo para llenar el registro
lbu     $t0,8($s3)      # carga rt con byte mem[8+($s3)];
                        # extensión cero para llenar el registro
sb      $t0,A($s3)      # almacenar byte 0 de rt a mem[A+($s3)]
```

En la figura 6.6 se muestra el formato de instrucción de máquina para las instrucciones *lb*, *lbu* y *sbu*. Como siempre, la dirección base se puede especificar como un valor absoluto o mediante un nombre simbólico.

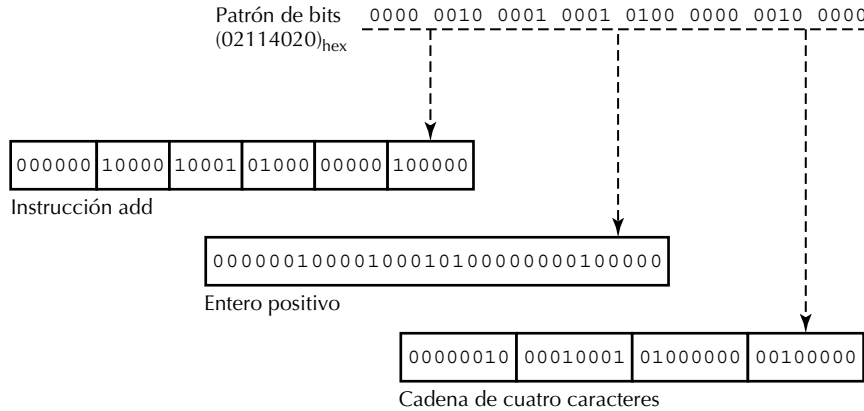


Figura 6.7 Una palabra de 32 bits no tiene significado inherente y se puede interpretar en un número de formas igualmente válidas en ausencia de otras pistas (por ejemplo, contexto) para el significado pretendido.

Esta sección concluye con una observación muy importante. Una cadena de bits, almacenada en memoria o en un registro, no tiene significado inherente. Una palabra de 32 bits puede significar diferentes cosas, dependiendo de cómo se interpreta o procesa. Por ejemplo, la figura 6.7 muestra que la misma palabra tiene tres significados diferentes, dependiendo de cómo se interpreta o usa. La palabra representada por el patrón hexa 0x02114020 se convierte en la instrucción add si se fetched de la memoria y ejecuta. La misma palabra, cuando se usa como un operando entero en una instrucción aritmética, representa un entero positivo. Finalmente, como una cadena de bits, la palabra puede representar una secuencia de cuatro símbolos ASCII. Ninguna de estas interpretaciones es más natural, o más válida, que otras. También son posibles otras interpretaciones. Por ejemplo, se puede ver la cadena de bits como un par de símbolos Unicode o el registro de asistencia para 32 estudiantes en una clase particular, donde 0 designa “ausente” y 1 significa “presente”.

6.5 Arreglos y apuntadores

En una amplia variedad de tareas de programación, se hace necesario caminar a través de un arreglo o lista, y examinar cada uno de sus elementos a la vez. Por ejemplo, para determinar el valor más grande en una lista de enteros, se debe examinar cada elemento de la lista. Hay dos formas básicas de lograr esto último:

1. *Índice*: Usar un registro que retenga el índice i e incremente el registro en cada paso para efectuar el movimiento del elemento i de la lista al elemento $i + 1$.
2. *Apuntador*: Usar un registro que apunta al (retiene la dirección de) elemento de lista que se examina y actualizarlo en cada paso para apuntar al siguiente elemento.

Cualquier enfoque es válido, pero el segundo es un poco más eficiente para MiniMIPS, dada su falta de un modo de direccionamiento indexado que permitiría usar el valor en el registro índice i en el cálculo de dirección. Para implementar el primer esquema, la dirección se debe calcular mediante muchas instrucciones que en esencia forman $4i$ y luego suman el resultado al registro que retiene la dirección base del arreglo o lista. En el segundo esquema, una sola instrucción que agrega 4 al apuntador de registro logra el avance hacia el siguiente elemento del arreglo. La figura 6.8 representa gráficamente los dos métodos.

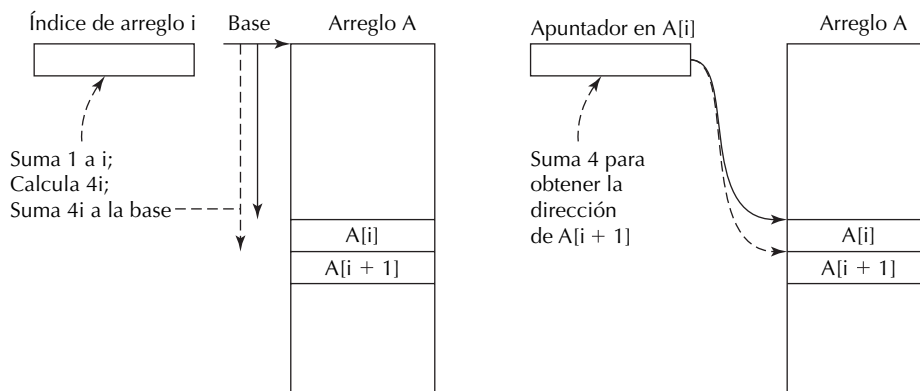


Figura 6.8 Uso de los métodos indexado y de actualización de puntero para caminar a través de los elementos de un arreglo.

Sin embargo, tratar con los punteros es conceptualmente más difícil y proclive al error, mientras que el uso de índices de arreglo aparece más natural y, por ende, es más fácil de entender. De este modo, usualmente no se recomienda el uso de punteros sólo para mejorar la eficiencia del programa. Por fortuna, los compiladores modernos son muy inteligentes como para sustituir tal código indexado con el código equivalente (más eficiente) que usan los punteros. Así, los programadores se pueden relajar y usar cualquier forma que sea más natural para ellos.

El uso de estos métodos se ilustra mediante dos ejemplos. El ejemplo 6.3 usa índices de arreglo, mientras que el ejemplo 6.4 obtiene ventaja de los punteros.

Ejemplo 6.3: Prefijo de suma máxima en una lista de enteros Considere una lista de enteros de longitud n . Un *prefijo* de longitud i para la lista dada consiste de los primeros enteros i en la lista, donde $0 \leq i \leq n$. Un prefijo de suma máxima, como el nombre implica, representa un prefijo para el que la suma de elementos es la más grande entre todos los prefijos. Por ejemplo, si la lista es $(2, -3, 2, 5, -4)$, su prefijo de suma máxima consiste de los primeros cuatro elementos y la suma asociada es $2 - 3 + 2 + 5 = 6$; ningún otro prefijo de la lista específica tiene una suma más grande. Escriba un programa MiniMIPS para encontrar la longitud del prefijo de suma máxima y la suma de sus elementos para una lista dada.

Solución: La estrategia para resolver este problema es más bien obvia. Comience por inicializar el prefijo max-sum a la longitud 0 con una suma de 0. Luego aumente gradualmente la longitud del prefijo, cada vez calculando la nueva suma y comparándola con la suma máxima hasta el momento. Cuando se encuentre una suma más grande, la longitud y la suma del prefijo max-sum se actualizan. Escriba el programa en la forma de un procedimiento que acepte la dirección base de arreglo A en \$a0 y su longitud n en \$a1, regrese la longitud del prefijo max-sum en \$v0 y la suma asociada en \$v1.

```

                                # base A en $a0, longitud n en $a1
mspfx: addi    $v0,$zero,0      # inicializa longitud en $v0 a 0
        addi    $v1,$zero,0      # inicializa max sum en $v1 a 0
        addi    $t0,$zero,0      # inicializa índice i en $t0 a 0
        addi    $t1,$zero,0      # inicializa suma que corre en $t1 a 0
loop:   add     $t2,$t0,$t0      # fija 2i en $t2
        add     $t2,$t2,$t2      # fija 4i en $t2

```

```

        addi    $t3,$t2,$a0    # fija 4i+A (dirección de A[i]) en $t3
        lw      $t4,0($t3)     # carga A[i] de mem[($t3)] en $t4
        add     $t1,$t1,$t4    # suma A[i] a suma que corre en $t1
        slt     $t5,$v1,$t1    # fija $t5 en 1 si máx sum < new sum
        bne     $t5,$zero,mdfy # si máx sum es menor, modificar
                                resultados
        j       test           # ¿hecho?
mdfy:    addi    $v0,$t0,1      # nuevo prefijo máx-sum tiene longitud i+1
        addi    $v1,$t1,0      # nueva suma máx es la suma que corre
test:    addi    $t0,$t0,1      # avanza el índice i
        slt     $t5,$t0,$a1    # fija $t5 en 1 si i < n
        bne     $t5,$zero,loop # repetir si i < n
done:    jr      $ra           # regresar length=($v0), máx sum=($v1)

```

Dado que los datos del prefijo se modifican sólo cuando se encuentra una suma estrictamente más grande, el procedimiento identifica el prefijo más corto para el caso de que existan muchos prefijos con la misma suma máxima.

Ejemplo 6.4: Clasificación de selección con el uso de un procedimiento de descubrimiento de máximo

Una lista dada de n números se puede clasificar en orden ascendente del modo siguiente. Encuentre un número más grande en la lista (puede haber más de uno) e intercámbielo con el último elemento en la lista. El nuevo último elemento está ahora en su posición correcta en orden clasificado. Ahora, clasifique los restantes $n - 1$ elementos con el mismo paso repetidamente. Cuando sólo queda un elemento, la clasificación está completa. Este método se conoce como *clasificación de selección*. Escriba un programa MiniMIPS para clasificación de selección con un procedimiento que encuentre el máximo elemento en una lista.

Solución: La figura 6.9 ilustra una iteración del algoritmo. La parte no clasificada de la lista se delimita mediante los dos apuntadores *first* y *last*. El procedimiento *máx* se llama para identificar un elemento más grande en la parte no clasificada junto con su ubicación en la lista. Entonces este elemento se intercambia con el último elemento de la lista, el apuntador *last* decrementa para apuntar al nuevo último elemento y el proceso se repite hasta que *first* = *last*.

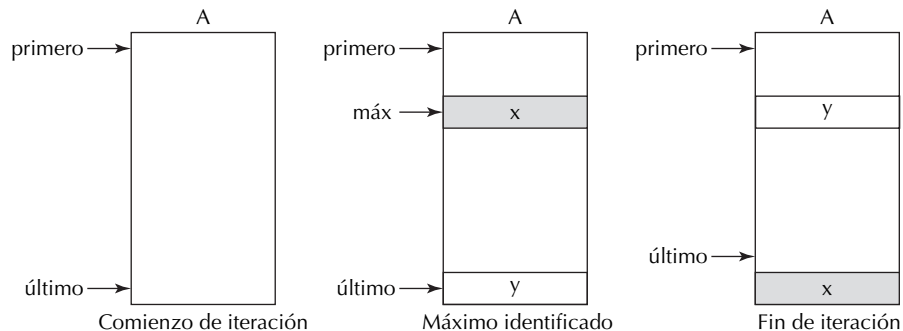


Figura 6.9 Una iteración de la clasificación de selección.

```

# uso de registro en programa de clasificación
#   $a0   apuntador a primer elemento en parte no clasificada
#   $a1   apuntador a último elemento en parte no clasificada

```

```

#   $t0   lugar temporal para valor de último elemento
#   $v0   apuntador a elemento máx en parte no clasificada
#   $v1   valor de elemento máx en parte no clasificada
sort: beq  $a0,$a1,done    # lista de elemento único clasificada
      jal  máx            # llamada del procedimiento máx
      lw   $t0,0($a1)     # carga último elemento en
      sw   $t0,0($v0)     # copia último elemento a ubicación máx
      sw   $v1,0($a1)     # copia valor máx a último elemento
      addi $a1,$a1,-4      # decrementa apuntador a último elemento
      j    sort           # repita clasificación para lista más
                           # pequeña
done: ...                 # continúa con resto del programa

# uso de registro en procedimiento máx
#   $a0   apuntador al primer elemento
#   $a1   apuntador al último elemento
#   $t0   apuntador al siguiente elemento
#   $t1   valor de siguiente elemento
#   $t2   resultado de comparación "(siguiente) < (máx)"
#   $v0   apuntador al elemento máx
#   $v1   valor de elemento máx

máx:  addi $v0,$a0,0      # inic apuntador máx a primer elemento
      lw   $v1,0($v0)     # inic valor máx a primer valor
      addi $t0,$a0,0      # inic apuntador siguiente a primero
loop:  beq  $t0,$a0,ret    # si siguiente = último, regreso
      addi $t0,$t0,4      # avance a elemento siguiente
      lw   $t1,0($t0)     # carga elemento siguiente en $t1
      slt  $t2,$t1,$v1    # ¿(siguiente) < (máx)?
      bne  $t2,$zero,loop # si (siguiente) < (máx), repetir con n/c
      addi $v0,$t0,0      # elemento siguiente es ahora elemento máx
      addi $v1,$t1,0      # valor siguiente es ahora valor máx
      j    loop           # cambio completo; ahora repetir
ret:   jr   $ra           # regreso a programa llamante

```

Puesto que el procedimiento máx sustituye el elemento máximo previamente identificado con uno último que tiene el mismo valor, el elemento máximo identificado es el último entre muchos valores iguales. Por ende, el algoritmo de clasificación preserva el orden original de muchos elementos iguales en la lista. Tal algoritmo de clasificación se denomina *algoritmo de clasificación estable*. Se sugiere comparar el procedimiento máx en este ejemplo con el programa del ejemplo 5.5, que, en esencia, realiza el mismo cálculo mediante direccionamiento indexado.

■ 6.6 Instrucciones adicionales

En el capítulo 5 se introdujeron 20 instrucciones para MiniMIPS (tabla 5.1) y cuatro instrucciones adicionales para llamada de procedimiento (jal) y datos orientados a bytes (lb, lbu, sb) hasta el momento en este capítulo. Como indican los ejemplos 6.3 y 6.4, se pueden escribir programas útiles y no triviales sólo con el uso de estas 24 instrucciones. Sin embargo, los MiniMIPS tienen instrucciones adicionales

que permiten la realización de cálculos más complejos y expresar programas de manera más eficiente. En esta sección se presentan algunas de estas instrucciones, ello lleva el total a 40 instrucciones. Éstas completan la parte central de la arquitectura de conjunto de instrucciones MiniMIPS. Todavía se tiene que cubrir las instrucciones que trata con aritmética de punto flotante y excepciones (es decir, los coprocesadores que se muestran en la figura 5.1). Las instrucciones de punto flotante se introducirán en el capítulo 12, mientras que las instrucciones relacionadas con el manejo de excepciones se introducirán según se necesiten, comenzando con la sección 14.6.

Se comienza por la presentación de muchas instrucciones aritméticas/lógicas adicionales. La instrucción multiplicar (*multiply*) (`mult`) representa una instrucción en formato R que coloca el producto doble palabra de los contenidos de dos registros en los registros Hi (mitad superior) y Lo (mitad inferior). La instrucción `div` calcula el resto y el cociente de los contenidos de dos registros fuente, y los coloca en los registros especiales Hi y Lo, respectivamente.

```
mult    $s0, $s1    # fija Hi,Lo en ($s0) × ($s1)
div     $s0, $s1    # fija Hi en ($s0) mod ($s1)
                        # y Lo en ($s0) / ($s1)
```

La figura 6.10 muestra las representaciones de máquina de las instrucciones `mult` y `div`.

Para ser capaz de usar los resultados de `mult` y `div`, se proporcionan dos instrucciones especiales, “mover de Hi” (`mfhi`) y “mover de Lo” (`mflo`), para copiar los contenidos de estos dos registros especiales en uno de los registros generales:

```
mfhi    $t0          # fija $t0 en (Hi)
mflo    $t0          # fija $t0 en (Lo)
```

La figura 6.11 muestra las representaciones de máquina de las instrucciones `mfhi` y `mflo`.

Ahora que se aprendió acerca de la instrucción multiplicar MiniMIPS, puede resultar atractivo usar `mult` para calcular el *offset* $4i$, que se necesita cuando se quiere acceder en memoria al elemento i -ésimo de un arreglo. Sin embargo, como se verá con más detalle en la parte tres de este libro, la multiplicación y la división constituyen operaciones más complejas y más lentas que la suma y la resta. Por ende, todavía es aconsejable calcular $4i$ mediante dos sumas ($i + i = 2i$ y $2i + 2i = 4i$) en lugar de una sola multiplicación.

Desde luego, una forma todavía más eficiente de calcular $4i$ es a través de un corrimiento izquierdo de dos bits de i , pues i es un entero sin signo (como usualmente es el caso en la indexación de arreglo). En la sección 10.5 se discutirá el corrimiento de valores signados. La instrucción MiniMIPS que permite el corrimiento izquierdo del contenido de un registro por una cantidad conocida se llama “corrimiento izquierdo lógico” (`sll`). De igual modo, “corrimiento derecho lógico” (`srl`) corre el contenido de un



Figura 6.10 Instrucciones multiplicar (`mult`) y dividir (`div`) de MiniMIPS.

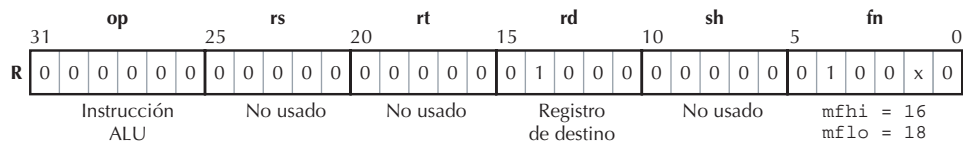


Figura 6.11 Instrucciones MiniMIPS para copiar los contenidos de los registros Hi y Lo en registros generales.



Figura 6.12 Las cuatro instrucciones de corrimiento lógico de MiniMIPS.

registro a la derecha. La necesidad de la calificación “lógica” se volverá clara conforme se discutan los corrimientos aritméticos en la sección 10.5. Por ahora, recuerde que los corrimientos lógicos mueven los bits de una palabra a la izquierda o a la derecha, y llenan las posiciones vacías con 0 y descartan cualquier bit que se mueva desde el otro extremo de la palabra. Las instrucciones `sll` y `srl` tratan con una cantidad de corrimiento constante que está dada en el campo “sh” de la instrucción. Para una cantidad de corrimiento variable específica en un registro, las instrucciones relevantes son `sllv` y `srlv`.

```

sll    $t0,$s1,2    # $t0 = ($s1) corrido izquierda por 2
srl    $t0,$s1,2    # $t0 = ($s1) corrido derecha por 2
sllv   $t0,$s1,$s0  # $t0 = ($s1) corrido izquierda por ($s0)
srlv   $t0,$s1,$s0  # $t0 = ($s1) corrido derecha por ($s0)

```

Un corrimiento izquierdo de h bits multiplica el valor sin signo x por 2^h , siempre que x sea muy pequeño como para que $2^h x$ todavía sea representable en 32 bits. Un corrimiento derecho de h bits divide un valor sin signo por 2^h , y descarta la parte fraccional y conserva el cociente entero. La figura 6.12 muestra la representación de máquina de las precedentes cuatro instrucciones de corrimiento.

Finalmente, en virtud de que los enteros sin signo son importantes en muchos cálculos, algunas instrucciones MiniMIPS tienen variaciones en las que uno o ambos operandos se consideran números sin signo. Estas instrucciones tienen los mismos nombres simbólicos que las correspondientes instrucciones con signo, pero con “u” agregado al final.

```

addu   $t0,$s0,$s1  # fijar $t0 en ($s0)+($s1)
subu   $t0,$s0,$s1  # fijar $t0 en ($s0)-($s1)
multu  $s0,$s1      # fijar Hi,Lo en ($s0)×($s1)
divu   $s0,$s1      # fijar Hi en ($s0)mod($s1) y
                   # Lo en ($s0)/($s1)
addiu  $t0,$s0,61   # fijar $t0 en ($s0)+61 (decimal);
                   # el operando inmediato se visualiza
                   # con signo (es extendido en signo)

```

Las representaciones de máquina de estas instrucciones son idénticas a las de las versiones con signo, excepto que el valor del campo `fn` es 1 más para instrucciones de tipo R y el valor del campo `op` es 1 más para `addiu` (figuras 5.5, 6.10 y 5.6).

Observe que el operando inmediato en `addiu` en realidad es un valor con signo de 16 bits que está extendido en signo a 32 bits antes de sumarse al operando sin signo en un registro especificado. Ésta es

una de las irregularidades que se deben recordar. Intuitivamente, uno puede pensar que ambos operandos de `addiu` son valores sin signo y esta expectativa es bastante lógica. Sin embargo, como consecuencia de que en MiniMIPS no existe instrucción “sustracción inmediata sin signo” y hay ocasiones cuando se necesita restar algo de un valor sin signo, se hace esta elección de diseño aparentemente ilógica.

En los capítulos 5 y 6 se introdujeron 37 instrucciones MiniMIPS. La tabla 6.2 resume estas instrucciones para revisión y fácil referencia. En este contexto, usted debe entender por completo la tabla

■ **TABLA 6.2** Las 40 instrucciones MiniMIPS tratadas en los capítulos 5-7.*

Clase	Instrucción	Uso	Significado	op	fn
Copia	Move from Hi	<code>mfhi rd</code>	$rd \leftarrow (Hi)$	0	16
	Move from Lo	<code>mflo rd</code>	$rd \leftarrow (Lo)$	0	18
	Load upper immediate	<code>lui rt, imm</code>	$rt \leftarrow (imm, 0x0000)$	15	
	Add	<code>add rd, rs, rt</code>	$rd \leftarrow (rs) + (rt)$; con desbordamiento	0	32
Aritmética	Add unsigned	<code>addu rd, rs, rt</code>	$rd \leftarrow (rs) + (rt)$; sin desbordamiento	0	33
	Subtract	<code>sub rd, rs, rt</code>	$rd \leftarrow (rs) - (rt)$; con desbordamiento	0	34
	Subtract unsigned	<code>subu rd, rs, rt</code>	$rd \leftarrow (rs) - (rt)$; sin desbordamiento	0	35
	Set less than	<code>slt rd, rs, rt</code>	$rd \leftarrow si\ (rs) < (rt)$ entonces 1 sino 0	0	42
	Multiply	<code>mult rs, rt</code>	Hi, Lo $\leftarrow (rs) \times (rt)$	0	24
	Multiply unsigned	<code>multu rs, rt</code>	Hi, Lo $\leftarrow (rs) \times (rt)$	0	25
	Divide	<code>div rs, rt</code>	Hi $\leftarrow (rs) \bmod (rt)$; Lo $\leftarrow (rs) \div (rt)$	0	26
	Divide unsigned	<code>divu rs, rt</code>	Hi $\leftarrow (rs) \bmod (rt)$; Lo $\leftarrow (rs) \div (rt)$	0	27
	Add immediate	<code>addi rt, rs, imm</code>	$rt \leftarrow (rs) + imm$; con desbordamiento	8	
	Add immediate unsigned	<code>addiu rt, rs, imm</code>	$rt \leftarrow (rs) + imm$; sin desbordamiento	9	
	Set less than immediate	<code>slti rd, rs, imm</code>	$rd \leftarrow si\ (rs) < imm$ entonces 1 sino 0	10	
	Shift left logical	<code>sll rd, rt, sh</code>	$rd \leftarrow (rt)$ corrimiento izquierdo por sh	0	0
Corrimiento	Shift right logical	<code>srl rd, rt, sh</code>	$rd \leftarrow (rt)$ corrimiento derecho por sh	0	2
	Shift right arithmetic	<code>sra rd, rt, sh</code>	Como srl, pero extendido en signo	0	3
	Shift left logical variable	<code>sllv rd, rt, rs</code>	$rd \leftarrow (rt)$ corrimiento izquierdo por (rs)	0	4
	Shift right logical variable	<code>srlv rd, rt, rs</code>	$rd \leftarrow (rt)$ corrimiento derecho por (rs)	0	6
	Shift right arith variable	<code>srav rd, rt, rs</code>	Como srlv, pero extendido en signo	0	7
	AND	<code>and rd, rs, rt</code>	$rd \leftarrow (rs) \wedge (rt)$	0	36
Lógica	OR	<code>or rd, rs, rt</code>	$rd \leftarrow (rs) \vee (rt)$	0	37
	XOR	<code>xor rd, rs, rt</code>	$rd \leftarrow (rs) \oplus (rt)$	0	38
	NOR	<code>nor rd, rs, rt</code>	$rd \leftarrow ((rs) \vee (rt))'$	0	39
	AND immediate	<code>andi rt, rs, imm</code>	$rt \leftarrow (rs) \wedge imm$	12	
	OR immediate	<code>ori rt, rs, imm</code>	$rt \leftarrow (rs) \vee imm$	13	
	XOR immediate	<code>xori rt, rs, imm</code>	$rt \leftarrow (rs) \oplus imm$	14	
	Load word	<code>lw rt, imm(rs)</code>	$rt \leftarrow mem[(rs) + imm]$	35	
Acceso a memoria	Load byte	<code>lb rt, imm(rs)</code>	Carga byte 0, extendido en signo	32	
	Load byte unsigned	<code>lbu rt, imm(rs)</code>	Carga byte 0, extendido cero	36	
	Store word	<code>sw rt, imm(rs)</code>	$mem[(rs) + imm] \leftarrow rt$	43	
	Store byte	<code>sb rt, imm(rs)</code>	Almacena byte 0	40	
	Jump	<code>j L</code>	ir a L	2	
Transferencia de control	Jump and link	<code>jal L</code>	ir a L; $\$31 \leftarrow (PC) + 4$	3	
	Jump register	<code>jr rs</code>	ir a (rs)	0	8
	Branch less than 0	<code>bltz rs, L</code>	si (rs) < 0 entonces ir a: L	1	
	Branch equal	<code>beq rs, rt, L</code>	si (rs) = (rt) entonces ir a: L	4	
	Branch not equal	<code>bne rs, rt, L</code>	si (rs) \neq (rt) entonces ir a: L	5	
	System call	<code>syscall</code>	Vea sección 7.6 (tabla 7.2)	0	12

*En este orden de ideas, también se incluyen las instrucciones corrimiento derecho aritmético (`sra`, `srav`), que se expondrán en la sección 10.5, y `syscall`.

6.2, excepto para las frases *set overflow* (fijar desbordamiento) y *no overflow* (sin desbordamiento) en la columna “Significado” y las siguientes tres instrucciones:

```
sra      $t0,$s1,2      # corrimiento derecho aritmético
sra     $t0,$s1,$s0     # corrimiento derecho aritmético variable
syscall                # llamada de sistema
```

Los corrimientos derechos aritméticos, y cómo difieren de los corrimientos derechos lógicos, se estudiarán en la sección 10.5. La llamada de sistema, incluido su uso para realizar operaciones entrada/salida, se discutirán en la sección 7.6, y sus detalles se presentarán en la tabla 7.2.

PROBLEMAS

6.1 Llamadas de procedimiento anidadas

Modifique el diagrama de llamada de procedimiento anidado en la figura 6.2 para corresponder a los cambios siguientes:

- El procedimiento *xyz* llama al nuevo procedimiento *def*.
- Después de llamar a *xyz*, el procedimiento *abc* llama a otro procedimiento *uvw*.
- El procedimiento *abc* llama a *xyz* dos veces diferentes.
- Después de llamar a *abc*, el programa principal llama a *xyz*.

6.2 Cómo elegir uno de tres enteros

Modifique el procedimiento del ejemplo 6.2 (encontrar el más grande de tres enteros) de modo que:

- También regrese el índice (0, 1, o 2) del valor más grande en *\$v1*.
- Encuentre el más pequeño de los tres valores en lugar del más grande
- Encuentre el intermedio de los tres valores en lugar del más grande

6.3 Divisibilidad por potencias de 2

Un entero binario que tiene *h* ceros consecutivos en su extremo derecho es divisible por 2^h . Escriba un procedimiento MiniMIPS que acepte un solo entero sin signo en el registro *\$a0* y regrese la potencia más grande de 2 por la que es divisible (un entero en [0, 32]) en el registro *\$v0*.

6.4 Distancia de Hamming entre dos palabras

La distancia de Hamming entre dos cadenas de bits de la misma longitud representa el número de posiciones en

la que las cadenas tienen diferentes valores de bits. Por ejemplo, la distancia de Hamming entre 1110 y 0101 es 3. Escriba un procedimiento que acepte dos palabras en los registros *\$a0* y *\$a1* y regrese sus distancias de Hamming (un entero en [0, 32]) en el registro *\$v0*.

6.5 Significados de una cadena de bits

- Decodifique la instrucción, el entero positivo y la cadena de cuatro caracteres ASCII de la figura 6.7.
- Repita la parte *a)* para el caso cuando la instrucción es *andi \$t1, \$s2, 13108*.
- Repita la parte *a)* para el caso cuando el entero es 825 240 373.
- Repita la parte *a)* para el caso cuando la cadena de caracteres es '*<US>*'.

6.6 Determinación de la longitud de una cadena de caracteres

Escriba un procedimiento, *howlong*, que lleve un apuntador a una cadena ASCII terminada en nulo en el registro *\$a0* y regrese la longitud de la cadena, con exclusión de su terminador nulo, en el registro *\$v0*. Por ejemplo, si el argumento para *howlong* apunta a la cadena '*not very long*,' el valor que regresa será el entero 13.

6.7 Desempacando una cadena de caracteres

Escriba una secuencia de instrucciones MiniMIPS (con comentarios) que separaría y colocaría los cuatro bytes de un operando específico en cadena de caracteres en el registro *\$s0* en los registros *\$t0* (byte menos significativo), *\$t2*, *\$t2* y *\$t3* (byte más significativo). Si es necesario, puede usar *\$t4* y *\$t4* como temporales.

6.8 Procedimiento recursivo para calcular $n!$

- La función $f(n) = n!$ se puede definir recursivamente como $f(n) = n \times f(n-1)$. Escriba un procedimiento recursivo, definido como un procedimiento que se llama a sí mismo, para aceptar n en $\$a0$ y regresa $n!$ en $\$v0$. Suponga n suficientemente pequeño como para que el valor de $n!$ encaje en el registro resultado.
- Repita la parte *a*), pero suponga que el resultado es una doble palabra y se regresa en los registros $\$v0$ y $\$v1$.

6.9 Prefijo, posfijo y sumas subsecuencias

Modifique el procedimiento en el ejemplo 6.3 (que encuentra el prefijo de suma máxima en una lista de enteros) de manera que:

- Identifique el prefijo más largo con suma máxima.
- Identifique el prefijo con suma mínima.
- Identifique el posfijo de suma máxima (elementos al final de la lista)
- Identifique la suma máxima subsecuencia. *Sugerencia:* Cualquier suma subsecuencia representa la diferencia entre dos sumas prefijo.

6.10 Ordenamiento de selección

Modifique el procedimiento del ejemplo 6.4 (ordenamiento de selección) de manera que:

- Ordene en forma descendente.
- Las claves que ordene en forma ascendente sean dobles palabras.

- Ordene la lista de claves en forma ascendente y reordene una lista de valores asociados con las claves de manera que, si una clave ocupa la posición j cuando la lista de clave se ordena, su valor asociado también está en la posición j de la lista reordenada de valores.

6.11 Cálculo de una suma de verificación XOR

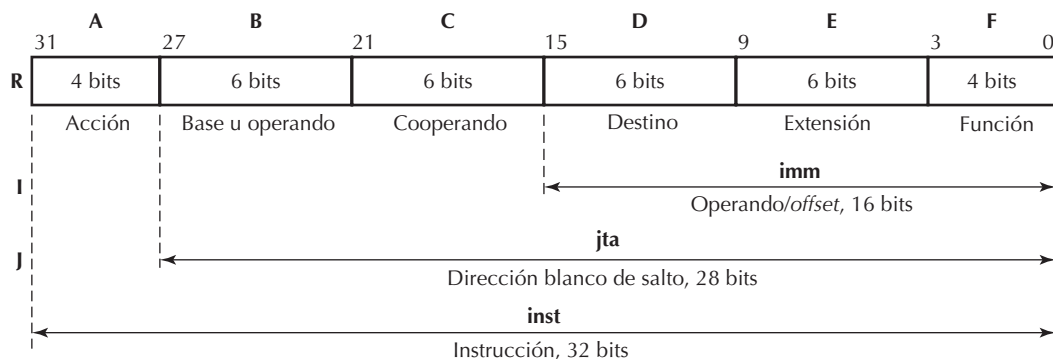
Escriba un procedimiento que acepte una cadena de bytes (dirección base en $\$a0$, longitud en $\$a1$) y regrese su suma de verificación (*checksum*) XOR, definida como la OR exclusiva de todos sus bytes, en $\$v0$.

6.12 Modificación del ISA MiniMIPS

Considere los siguientes formatos de instrucción de 32 bits que son muy similares a los de MiniMIPS. Los seis campos son diferentes en sus anchos y nombres, pero tienen similar importancia que los campos correspondientes de los MiniMIPS.

Responda cada una de las siguientes preguntas bajo dos diferentes suposiciones: 1) el ancho del registro permanece en 32 bits y 2) los registros tienen 64 bits de ancho. Suponga que el campo *action* retiene 00xx para instrucciones en formato R, 1xxx para formato I y 01xx para formato J.

- ¿Cuál es el máximo número posible de instrucciones diferentes en cada una de las tres clases R, I y J?
- ¿Se pueden codificar todas las instrucciones que se muestran en la tabla 6.2 con este nuevo formato?
- ¿Puede pensar en alguna nueva instrucción deseable de agregar como consecuencia de los cambios?



6.13 Llamada a un procedimiento

Con el uso del procedimiento del ejemplo 6.1, escriba una secuencia de instrucciones MiniMIPS para encontrar el elemento de una lista de enteros con el valor absoluto más grande. Suponga que la dirección de comienzo y el número de enteros en la lista se proporcionan en los registros `$s0` y `$s1`, respectivamente.

6.14 Pila y apuntadores de marco

Dibuje un diagrama esquemático de la *pila* (similar a la figura 6.5), correspondiente a las relaciones de llamado que se muestran en la figura 6.2, para cada uno de los casos siguientes, que muestre la *pila* y los apuntadores de marco como flechas:

- Dentro del programa principal, antes de llamar al procedimiento `abc`
- Dentro del procedimiento `abc`, antes de llamar al procedimiento `xyz`
- Dentro del procedimiento `xyz`
- Dentro del procedimiento `abc`, después de regresar del procedimiento `xyz`
- Dentro del programa principal, después de regresar del procedimiento `abc`

6.15 Conversión ASCII/entero

- Un número decimal se almacena en memoria en la forma de una cadena ASCII terminada en nulo. Escriba un procedimiento MiniMIPS que acepte la dirección de comienzo de la cadena ASCII en el re-

gistro `$a0` y regresa el valor entero equivalente en el registro `$v0`. Ignore la posibilidad de desbordamiento y suponga que el número puede comenzar con un dígito o un signo (+ o -).

- Escriba un procedimiento MiniMIPS para realizar el inverso de la conversión de la parte *a*).

6.16 Emulación de instrucciones corrimiento

Demuestre cómo el efecto de cada una de las siguientes instrucciones se puede lograr mediante el uso sólo de instrucciones vistas antes de la sección 6.5

- Corrimiento izquierdo lógico `sll`.
- Corrimiento izquierdo lógico variable `sllv`.
- Corrimiento derecho lógico `srl`. *Sugerencia:* Use rotación izquierda (corrimiento circular) por una cantidad adecuada seguida por ceros en algunos de los bits.
- Corrimiento derecho lógico variable `srlv`.

6.17 Emulación de otras instrucciones

- Identifique todas las instrucciones de la tabla 6.2, cuyo efecto se pueda lograr mediante no más de otras dos instrucciones. Use cuando mucho un registro para valores intermedios (use `$at` si fuera necesario).
- ¿Existen instrucciones en la tabla 6.2 cuyos efectos no se pueden lograr a través de otras instrucciones, sin importar cuántas instrucciones o registros se permita usar?

REFERENCIAS Y LECTURAS SUGERIDAS

- | | |
|----------|--|
| [Kane92] | Kane, G. y J. Heinrich, <i>MIPS RISC Architecture</i> , Prentice Hall, 1992. |
| [MIPS] | Sitio Web de tecnologías MIPS. Siga las ligas de arquitectura y documentación en http://www.mips.com/ |
| [Patt98] | Patterson, D. A. y J. L. Hennessy, <i>Computer Organization and Design: The Hardware/Software Interface</i> , Morgan Kaufmann, 2a. ed., 1998. |
| [Sail96] | Sailer, P. M. y D. R. Kaeli, <i>The DLX Instruction Set Architecture Handbook</i> , Morgan Kaufmann, 1996. |
| [SPIM] | SPIM, simulador de software de descarga gratuita para un subconjunto del conjunto de instrucciones MIPS; más información en la sección 7.6.
http://www.cs.wisc.edu/~larus/spim.html |
| [Swee99] | Sweetman, D., <i>See MIPS Run</i> , Morgan Kaufmann, 1999. |

PROGRAMAS EN LENGUAJE ENSAMBLADOR

“El código dice qué hace el programa, las observaciones (comentarios) dicen lo que se supone está haciendo. Depurar es reconciliar las diferencias.”

Tony Amico, Tonyism

“El software se encuentra entre el usuario y la máquina.”

Harlan D. Mills

TEMAS DEL CAPÍTULO

- 7.1 Lenguajes de máquina y ensamblador
- 7.2 Directivas de ensamblador
- 7.3 Seudoinstrucciones
- 7.4 Macroinstrucciones
- 7.5 Ligado y cargado
- 7.6 Corrida de programas ensamblador

Para construir y correr programas eficientes y útiles en lenguaje ensamblador, se necesita más de un conocimiento de los tipos y formatos de instrucciones de máquina. Por un lado, el ensamblador necesita ciertos tipos de información acerca del programa y sus datos que no son evidentes a partir de las instrucciones mismas. Asimismo, la conveniencia señala el uso de ciertas pseudoinstrucciones y macros, que, aunque no están en correspondencia uno a uno con las instrucciones de máquina, se convierten fácilmente a las últimas por el ensamblador. Para redondear los temas, en este capítulo se presentan breves descripciones de los procesos de carga (*loading*) y ligado (*linking*) y las ejecuciones de programas en lenguaje ensamblador.

■ 7.1 Lenguajes de máquina y ensamblador

Las instrucciones de máquina se representan como cadenas de bits binarios. Para el caso de MiniMIPS, todas las instrucciones tienen un ancho uniforme de 32 bits y, por ende, encajan en una sola palabra. En el capítulo 8 se verá que ciertas máquinas usan instrucciones de ancho variable, ello lleva a una codificación más eficiente de las acciones pretendidas. Ya sea de ancho fijo o variable, el *código de máquina* binario, o su equivalente representación hexa, es poco conveniente para la comprensión humana. De hecho, por esta razón se desarrollaron *lenguajes ensambladores* que permiten el uso de nombres

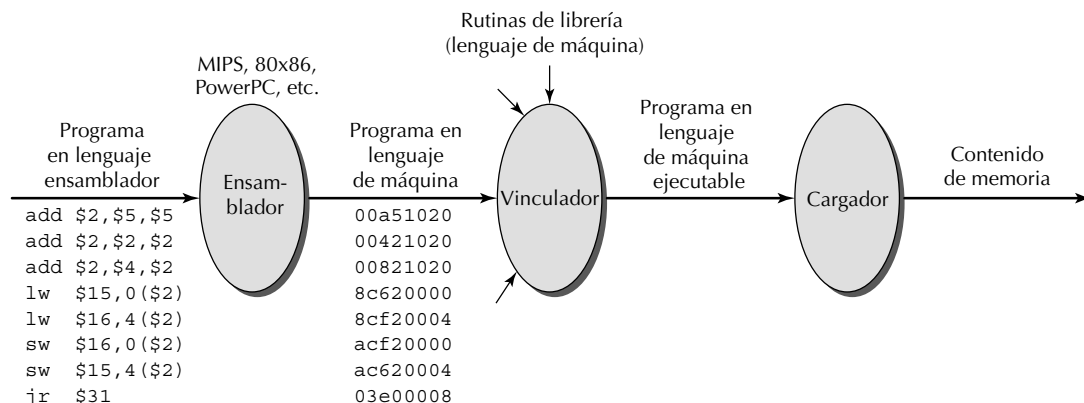


Figura 7.1 Pasos para transformar un programa en lenguaje ensamblador a un programa ejecutable residente en memoria.

simbólicos para instrucciones y sus operandos. Además, los ensambladores aceptan números en una diversidad de representaciones simples y naturales y los convierten a formatos de máquina requeridos. Finalmente, los ensambladores permiten el uso de *seudoinstrucciones* y *macros* que sirven al mismo propósito como abreviaturas y acrónimos en lenguajes naturales, esto es, permitir representaciones más compactas que son más fáciles de escribir, leer y entender.

En los capítulos 5 y 6 se introdujeron partes del lenguaje ensamblador para la hipotética computadora MiniMIPS. Las secuencias de instrucción escritas en lenguaje ensamblador simbólico se traducen a lenguaje de máquina mediante un programa llamado *ensamblador*. Múltiples módulos de programa que se ensamblan de manera independiente se *vinculan* en conjunto y combinan con rutinas de *librería* (predesarrolladas) para formar un *programa ejecutable* completo que luego se *carga* en la memoria de una computadora (figura 7.1). Los módulos de programa que se ensamblan por separado y luego se vinculan resulta benéfico porque cuando un módulo se modifica, las otras partes del programa no tienen que reensamblarse. En la sección 7.5 se discutirá con más detalle el trabajo del vinculador (*linker*) y el cargador (*loader*).

En vez de cargarse en la memoria de una computadora, las instrucciones de un programa en lenguaje de máquina se pueden *interpretar* mediante un *simulador*: un programa que examina cada una de las instrucciones y realiza su función, y que actualiza variables y estructuras de datos que corresponden a registros y otras partes de máquina que retienen información. Tales simuladores se usan durante el proceso de diseño de nuevas máquinas para escribir y depurar programas antes de que cualquier hardware funcional esté disponible.

La mayoría de las instrucciones en lenguaje ensamblador están en correspondencia uno a uno con las instrucciones de máquina. El proceso de traducir tal instrucción en lenguaje ensamblador en la correspondiente instrucción de máquina, involucra la asignación de códigos binarios apropiados para especificar simbólicamente operaciones, registros, localidades de memoria, operandos inmediatos, etc. Algunas pseudoinstrucciones y todas las macroinstrucciones corresponden a múltiples instrucciones de máquina. Un ensamblador lee un *archivo fuente* que contiene el programa en lenguaje ensamblador y la información que lo acompaña (directivas de ensamblador o ciertos detalles de contabilidad) y en el proceso de producir el correspondiente programa en lenguaje de máquina:

Conserva una *tabla de símbolos* que contienen correspondencias nombre-dirección.

Construye los segmentos de texto y datos del programa.

Forma un *archivo objeto* que contiene encabezado, texto, datos e información de reubicación.

El proceso ensamblador se realiza en dos pasos. La función principal del primero es la construcción de la tabla de símbolos. Un *símbolo* constituye una cadena de caracteres que se usan como etiqueta de instrucción o como nombre de variable. Conforme se leen las instrucciones, el ensamblador mantiene un *contador de localidad de instrucción* que determina la posición relativa de la instrucción en el programa de lenguaje de máquina y, por tanto, el equivalente numérico de la etiqueta de la instrucción, si hubiere alguna. Lo anterior se realiza suponiendo que el programa se cargará al comienzo con la dirección 0 en la memoria. Puesto que la localidad eventual del programa usualmente es diferente, el ensamblador también compila información acerca de lo que se necesita hacer si el programa se carga comenzando con otra dirección (arbitraria). Esta *información de reubicación* forma parte del archivo producido por el ensamblador y lo usará el cargador para modificar el programa de acuerdo con su localidad eventual en la memoria.

Considere, por ejemplo, la instrucción en lenguaje ensamblador:

```
test:    bne    $t0, $s0, done
```

Mientras el ensamblador lee esta línea, detecta el símbolo de operación “bne”, dos símbolos de registro “\$t0” y “\$s0”, y las etiquetas de instrucción “test” y “done”. El código adecuado para “bne” se lee desde una tabla de *opcode* (que también contiene información acerca del número y tipos de operandos que se esperan), mientras que los significados de “\$t0” y “\$s0” se obtienen de otra tabla de referencias. El símbolo “test” se ingresa en la tabla de símbolos junto con el contenido actual del contador de localidad de la instrucción como su valor. El símbolo “done” puede o no estar en la tabla de símbolos, dependiendo de si la bifurcación se orienta hacia atrás (hacia una instrucción previa) o hacia adelante. Si se trata de una bifurcación hacia atrás, entonces el equivalente numérico de “done” ya se conoce y se puede usar en su lugar. Para el caso de una bifurcación hacia adelante, se crea una entrada en la tabla de símbolos para “done” con su valor en blanco; el valor se llenará en la segunda pasada después de que a todas las etiquetas de instrucción se les hayan asignado equivalentes numéricos.

Este proceso de *resolver referencias hacia adelante* representa la razón principal por la que se usa una segunda pasada. También es posible realizar todo el proceso en una sola pasada, pero esto último requeriría dejar marcadores adecuados para las referencias no resueltas por adelantado, de modo que sus valores se puedan llenar al final.

La figura 7.2 muestra un programa corto en lenguaje ensamblador, su versión ensamblada y la tabla de símbolos que se creó durante el proceso de ensamblado. El programa está incompleto y no tiene sentido, pero sí ilustra algunos de los conceptos apenas discutidos.

Programa en lenguaje ensamblador	Localidad	Programa en lenguaje de máquina						
addi \$s0,\$zero,9	0	001000	00000	10000	00000	00000	001001	
sub \$t0,\$s0,\$s0	4	000000	10000	10000	01000	00000	100010	
add \$t1,\$zero,\$zero	8	000000	01001	00000	00000	00000	100000	
test: bne \$t0,\$s0,done	12	000101	01000	10000	00000	00000	001100	
addi \$t0,\$t0,1	16	001000	01000	01000	00000	00000	000001	
add \$t1,\$s0,\$zero	20	000000	10000	00000	01001	00000	100000	
j test	24	000010	00000	00000	00000	00000	000011	
done: sw \$t1,result(\$gp)	28	101011	11100	01001	00000	00011	111000	
		op	rs	rt	rd	sh	fn	

done	28
result	248
test	12

Tabla de símbolos

Las fronteras de campo se muestran para facilitar la comprensión

Determinado a partir de directivas de ensamblador que no se muestran aquí

Figura 7.2 Programa en lenguaje ensamblador, su versión en lenguaje de máquina y la tabla de símbolos creada durante el proceso de ensamblado.

■ 7.2 Directivas de ensamblador

Las directivas de ensamblador proporcionan al ensamblador información acerca de cómo traducir el programa, pero en sí mismas no llevan a la generación de las correspondientes instrucciones de máquina. Por ejemplo, las directivas de ensamblador pueden especificar la plantilla de datos en el segmento de datos del programa, o definir variables con nombres simbólicos y valores iniciales deseados. El ensamblador lee estas directivas y los toma en cuenta al procesar el resto de las líneas del programa.

En convención MiniMIPS, las directivas de ensamblador comienzan con un punto para distinguirlas de las instrucciones. A continuación se presenta una lista de directivas de ensamblador MiniMIPS:

```
.macro           # iniciar macro (ver sección 7.4)
.end_macro      # fin de macro (ver sección 7.4)
.text           # inicia segmento de texto de programa
...            # el texto del programa va aquí
.data           # inicia segmento de datos del programa
tiny: .byte     156,0x7a # nombra e inicializa byte(s) de datos
max:  .word     35000    # nombra e inicializa palabra(s) de datos
small: .float    2E-3    # nombra short float (ver capítulo 12)
big:   .double   2E-3    # nombra long float (ver capítulo 12)
      .align    2        # alinea siguiente objeto en frontera de
                        palabra
array: .space    600     # reserva 600 bytes = 150 palabras
str1:  .ascii    "a*b"   # nombra e inicializa cadena ASCII
str2:  .asciiz   "xyz"   # cadena ASCII con terminación nula
      .global   main     # considera main un nombre global
```

Las directivas “macro” y “end_macro” marcan el comienzo y fin de la definición para una macroinstrucción; se describirán en la sección 7.4 y aquí aparecen sólo para efectos de la lista. Las directivas “text” y “.data” señalan el comienzo de los segmentos de texto y datos de un programa. Por ende, “text” indica al ensamblador que las líneas subsiguientes se deben interpretar como instrucciones. Dentro del segmento de texto, las instrucciones y pseudoinstrucciones (ver sección 7.3) se pueden usar a voluntad.

En el segmento de datos, a los valores de datos se les puede asignar espacio de memoria, dar nombres simbólicos e inicializar a valores deseados. La directiva de ensamblador “.byte” se usa para definir uno o más elementos de datos con tamaño de byte con los valores iniciales deseados y para asignar un nombre simbólico al primero. En el ejemplo “tiny: .byte 156, 0x7a”, se definen un byte que retiene 156 seguido por un segundo byte que contiene el valor hexa 0x7a (122 decimal), y al primero se le da el nombre simbólico “tiny”. En el texto del programa, una dirección de la forma tiny (\$s0) hará referencia al primero o segundo de estos bytes si \$s0 retiene 0 o 1, respectivamente. Las directivas “.word” y “.float”, así como “.double” son similares a “.byte”, excepto que definen palabras (por ejemplo, enteros sin signo o con signo), números punto flotante cortos y números punto flotante largos, respectivamente.

La directiva “.align h” se usa para forzar a la dirección del siguiente objeto de datos a ser un múltiplo de 2^h (alineado con frontera de 2^h bytes). Por ejemplo, “.align 2” fuerza a la dirección siguiente a ser múltiplo de $2^2 = 4$; por tanto, a alinear un elemento de datos con tamaño de palabra con la llamada frontera de palabra. La directiva “.space” se usa para reservar un número especificado de bytes en el segmento de datos, usualmente para ofrecer espacio de almacenamiento para arreglos. Por ejemplo, “array: .space 600” reserva 600 bytes de espacio, que entonces se pueden usar para almacenar una cadena de bytes de longitud 600, un vector o longitud de palabra de 150, o un vector de 75 elementos de

valores punto flotante largos. El tipo de uso no se especifica en la directiva en sí, pero será una función de cuáles valores se coloquen ahí y cuáles instrucciones se apliquen a ellos. Preceder esta directiva con “.align 2” o “.align 3” permite usar el arreglo para palabras o dobles palabras, respectivamente. Observe que los arreglos usualmente no son inicializados por las directivas en el segmento de datos, sino más bien por instrucciones *store* explícitas, como parte de la ejecución del programa.

Las cadenas de caracteres ASCII se podrían definir con el uso de la directiva “.byte”. Sin embargo, con el propósito de evitar obtener y mencionar el código numérico ASCII para cada carácter y mejorar la legibilidad, se ofrecen dos directivas especiales. La directiva “.ascii” define una secuencia de bytes que mantienen los códigos ASCII para los caracteres que se mencionan entre comillas. Por ejemplo, los códigos ASCII para “a”, “*” y “b” se pueden colocar en tres bytes consecutivos y a la cadena se le da el nombre simbólico “str1” a través de la directiva “str1: .ascii “a*b””. Es práctica común terminar una cadena ASCII con el símbolo especial *null* (que tiene el código ASCII 0x00); de esta forma, el final de la cadena simbólica se reconoce fácilmente. Por tanto, la directiva “.asciiz” prueba tener el mismo efecto que “.ascii”, excepto que adiciona el símbolo *null* a la cadena y, por tanto, crea una cadena con un símbolo adicional al final.

Finalmente, la directiva “.global” define uno o más nombres como teniendo significancia en el sentido de que se les puede hacer referencia desde otros archivos.

El ejemplo 7.1 muestra el uso de algunas de estas directivas en el contexto de un programa completo en lenguaje ensamblador. Todos los ejemplos de programación subsecuentes en este capítulo usarán directivas de ensamblador.

Ejemplo 7.1: Composición de directivas de ensamblador simples Escribe una directiva de ensamblador para lograr cada uno de los siguientes objetivos:

- Almacenar el mensaje de error “Advertencia: ¡La impresora no tiene papel! En memoria.
- Establecer una constante llamada “size” con valor 4.
- Establecer una variable entera llamada “width” e inicializarla a 4.
- Establecer una constante llamada “mill” con el valor 1000000 (un millón).
- Reservar espacio para un vector entero “vect” de longitud 250.

Solución:

- `noppr: .asciiz "Advertencia: ¡La impresora no tiene papel!"`
- `size: .byte 4 # constante pequeña encaja en un byte`
- `width: .word 4 # byte podría ser suficiente, pero ...`
- `mill: .word 1000000 # constante demasiado grande para byte`
- `vect: .space 1000 # 250 palabras = 1000 bytes`

Para la parte a), se especifica una cadena ASCII terminada en *null* porque, con el terminador *null*, no se necesita conocer la longitud de la cadena durante su uso; si se decidiera cambiar la cadena, sólo esta directiva tendrá que modificarse. Para la parte c), en ausencia de información acerca del rango de width, se le asigna una palabra de almacenamiento.

7.3 Seudoinstrucciones

Aunque cualquier cálculo o decisión se puede moldear en términos de las instrucciones MiniMIPS simples cubiertas hasta el momento, a veces es más sencillo y más natural usar formulaciones alternas. Las *seudoinstrucciones* permiten formular cálculos y decisiones en formas alternas no directamente apoyadas por hardware. El ensamblador MiniMIPS tiene cuidado de traducir éstas a instrucciones básicas

soportadas por hardware. Por ejemplo, MiniMIPS carece de una instrucción NOT lógica que provocaría que todos los bits de una palabra se invirtieran. Aun cuando el mismo efecto se puede lograr por

```
nor    $s0,$s0,$ ero    # complemento ($s0)
```

sería mucho más natural, y más fácilmente comprensible, si se pudiera escribir:

```
not    $s0                # complemento ($s0)
```

Por tanto, “not” se define como una seudoinstrucción de MiniMIPS; se reconoce a esta última como instrucciones ordinarias para el ensamblador MiniMIPS y se sustituye por la instrucción equivalente “nor” antes de su conversión a lenguaje de máquina.

La seudoinstrucción “not” apenas definida traduce a una sola instrucción en lenguaje de máquina MiniMIPS. Algunas seudoinstrucciones se deben sustituir con más de una instrucción y pueden involucrar valores intermedios que se deben almacenar. Por esta razón, el registro \$1 se dedica al uso del ensamblador y se le da el nombre simbólico \$at (temporal ensamblador, figura 5.2). Es evidente que el ensamblador no puede suponer la disponibilidad de otros registros; en consecuencia, debe limitar sus valores intermedios a \$at y, quizás, el registro de destino en la seudoinstrucción misma. Un ejemplo es la seudoinstrucción “abs” que coloca el valor absoluto del contenido de un registro fuente en un registro de destino:

```
abs    $t0,$s0            # coloca |($s0)| en $t0
```

El ensamblador MiniMIPS puede traducir esta seudoinstrucción en la siguiente secuencia de cuatro instrucciones MiniMIPS:

```
add    $t0,$s0,$ ero      # copia el operando x en $t0
slt    $at,$t0,$ ero      # ¿es x negativo?
beq    $at,$ ero,+4        # si no, saltar la siguiente instrucción
sub    $t0,$ ero,$s0       # el resultado es 0 - x
```

La tabla 7.1 contiene una lista completa de seudoinstrucciones para el ensamblador MiniMIPS. Las seudoinstrucciones NOT aritmética y lógica son fáciles de entender. Las instrucciones *rotate* (rotar) son como corrimientos, excepto que los bits que salen de un extremo de la palabra se reinsertan en el otro extremo. Por ejemplo, el patrón de bits 01000110, cuando se rota a la izquierda (*left-rotated*) por dos bits, se convierte en 00011001; esto es, los dos bits 01 corridos desde el extremo izquierdo ocupan las dos posiciones menos significativas en el resultado rotado a la izquierda. La seudoinstrucción *load immediate* (carga inmediata) permite que cualquier valor inmediato se coloque en un registro, incluso si el valor no encaja en 16 bits. La seudoinstrucción *load address* (cargar dirección) es bastante útil para inicializar apuntadores en registros. Suponga que se quiere tener un apuntador para auxiliar el tránsito a través de un arreglo. Este apuntador se debe inicializar para apuntar al primer elemento del arreglo. Lo anterior no se puede hacer mediante instrucciones MiniMIPS regulares. La instrucción `lw $s0,array` colocaría el primer elemento del arreglo, no su dirección, en el registro \$s0, mientras que la seudoinstrucción `la $s0,array` coloca el equivalente numérico de la etiqueta “array” en \$s0. Las restantes seudoinstrucciones *load/store* y *branch-related* (relacionadas con bifurcación) también son comprensibles.

Debido a que definir seudoinstrucciones constituye una buena forma de practicar programación en lenguaje ensamblador, los problemas al final de este capítulo hipotetizan otras seudoinstrucciones y le piden escribir las correspondientes instrucciones MiniMIPS que debe producir un ensamblador. El ejemplo 7.2 presenta como modelo los lenguajes de máquina equivalentes de algunas de las seudoinstrucciones de la tabla 7.1. El ejemplo 7.3 muestra cómo las directivas de ensamblador y las seudoinstrucciones son útiles para crear programas completos.

■ **TABLA 7.1** Seudoinstrucciones aceptadas por el ensamblador MiniMIPS.

Clase	Seudoinstrucción	Uso	Significado
Copia	Move	move regd, regs	regd \leftarrow (regs)
	Load address	la regd, address	carga dirección calculada, no contenido
	Load immediate	li regd, anyimm	regd \leftarrow valor inmediato arbitrario
Aritmética	Absolute value	abs regd, regs	regd \leftarrow (regs)
	Negate	neg regd, regs	regd \leftarrow -(regs); con desbordamiento
	Multiply (into register)	mul regd, reg1, reg2	regd \leftarrow (reg1) \times (reg2); sin desbordamiento
	Divide (into register)	div regd, reg1, reg2	regd \leftarrow (reg1) \div (reg2); con desbordamiento
	Remainder	rem regd, reg1, reg2	regd \leftarrow (reg1) mod (reg2)
	Set greater than	sgt regd, reg1, reg2	regd \leftarrow si (reg1) > (reg2) entonces 1 de otro modo 0
	Set less or equal	sle regd, reg1, reg2	regd \leftarrow si (reg1) \leq (reg2) entonces 1 de otro modo 0
Corrimiento	Set greater or equal	sge regd, reg1, reg2	regd \leftarrow si (reg1) \geq (reg2) entonces 1 de otro modo 0
	Rotate left	rol regd, reg1, reg2	regd \leftarrow (reg1) rotado izquierdo por (reg2)
	Rotate right	ror regd, reg1, reg2	regd \leftarrow (reg1) rotado derecho por (reg2)
Lógica	NOT	not reg	reg \leftarrow (reg)'
Acceso a memoria	Load doubleword	ld regd, address	carga <i>regd</i> y el registro siguiente
	Store doubleword	sd regd, address	almacena <i>regd</i> y el registro siguiente
Transferencia de control	Branch less than	blt reg1, reg2, L	si (reg1) < (reg2) entonces ir a L
	Branch greater than	bgt reg1, reg2, L	si (reg1) > (reg2) entonces ir a L
	Branch less or equal	ble reg1, reg2, L	si (reg1) \leq (reg2) entonces ir a L
	Branch greater or equal	bge reg1, reg2, L	si (reg1) \geq (reg2) entonces ir a L

Ejemplo 7.2: Seudoinstrucciones MiniMIPS Para cada una de lasseudoinstrucciones siguientes definidas en la tabla 7.1, escriba las instrucciones correspondientes producidas por el ensamblador MiniMIPS.

```
parta: neg    $t0,$s0        # $t0 = -($s0); establece desbordamiento
partb: rem    $t0,$s0,$s1    # $t0 = ($s0) mod ($s1)
partc: li     $t0,imm        # $t0 = valor inmediato arbitrario
partd: blt    $s0,$s1,label  # si ($s0) < ($s1), ir a label
```

Solución

```
parta: sub    $t0,$ero,$s0   # -($s0) = 0 - ($s0)
partb: div    $s0,$s1        # dividir; el resto está en Hi
      mfhi    $t0            # copia (Hi) en $t0
partc: addi   $t0,$ero,imm    # si imm encaja en 16 bits
partc: lui    $t0,upperhalf   # si imm necesita 32 bits, el ensamblador
      ori     $t0,lowerhalf   # debe extraer mitades superior e inferior
partd: slt    $at,$s0,$s1     # ¿($s0) < ($s1)?
      bne     $at,$ero,label  # si ($s0) < ($s1), ir a
```

Observe que, para la parte c), el lenguaje mecánico equivalente de laseudoinstrucción difiere dependiendo del tamaño del operando inmediato proporcionado.

Ejemplo 7.3: Formación de programas en lenguaje ensamblador completos Agregue directivas de ensamblador y haga otras modificaciones necesarias en el programa parcial del ejemplo 6.4 (clasificación de selección con el uso de un procedimiento de búsqueda de máximo) para convertirlo en un programa completo, excluyendo las entradas y salidas de datos.

Solución: El programa parcial del ejemplo 6.4 supone una lista de palabras de entrada, con apuntadores hacia sus primero y último elementos disponibles en los registros \$a0 y \$a1. La lista clasificada ocupará las mismas localidades de memoria que la lista original (*in-place sorting*: clasificación en el lugar).

```

.global main          # considera main un nombre global
.data                 # inicia segmento de datos de programa
size: .word 0         # espacio para tamaño de lista en palabras
list: .space 4000     # supone que lista tiene hasta 1000
                      # elementos
.text                 # inicia segmento de texto de programa
main: ...             # entrada size y list
    la    $a0,list    # inic apuntador al primer elemento
    lw    $a1,size    # coloca size en $a1
    addi  $a1,$a1,-1   # offset en palabras al último elemento
    sll   $a1,$a1,2    # offset en bytes a último elemento
    add   $a1,$a0,$a1  # inic apuntador a último elemento
sort: ...             # resto del programa del ejemplo 6.4
done: ...             # list clasificada de salida

```

Observe que el programa completo se llama “main” e incluye el programa “sort” del ejemplo 6.4, incluido su procedimiento “máx”.

7.4 Macroinstrucciones

Una *macroinstrucción* (*macro*, para abreviar) constituye un mecanismo para denominar a una secuencia de instrucciones usada con frecuencia para evitar que se especifique la secuencia por completo cada vez. Como analogía, se puede escribir o teclear “IEC” en un documento borrador, con lo que se instruye al mecanógrafo o software procesador de palabra a sustituir todas las ocurrencias de “IEC” en la versión final con el nombre completo “Departamento de Ingeniería Eléctrica y de Computación”. Como consecuencia de que la misma secuencia de instrucciones puede involucrar diferentes especificaciones de operando (por ejemplo, registros) cada vez que se usen, una macro tiene un conjunto de parámetros formales que se sustituyen con parámetros reales durante el proceso de ensamblado. Las macros están delimitadas por directivas de ensamblador especiales:

```

.macro name(arg list) # nombre de macro y argumento(s)
...                  # instrucciones que definen la macro
.endmacro            # terminador de macro

```

En este punto pueden surgir dos preguntas naturales:

1. ¿En qué es diferente una macro de una seudoinstrucción?
2. ¿En qué es diferente una macro de un procedimiento?

Las seudoinstrucciones se incorporan en el diseño de un ensamblador y, por tanto, están fijas para el usuario; en este contexto, las macros son definidas por el usuario. Adicionalmente, una seudoinstruc-

ción parece exactamente como una instrucción, mientras que una macro parece más como un procedimiento en un lenguaje de alto nivel; por ejemplo, si usted no sabe que “move” no representa una instrucción en lenguaje de máquina de MiniMIPS, no podría decirlo por su apariencia. En cuanto a las diferencias con un procedimiento, se usan al menos dos instrucciones *jump* para llamar y regresar de un procedimiento, mientras que una macro es sólo una notación abreviada para varias instrucciones en lenguaje ensamblador; después de que la macro se sustituye por sus instrucciones equivalentes en el programa no quedan trazas de la macro.

El ejemplo 7.4 proporciona la definición y el uso de una macro completa. Es instructivo comparar los ejemplos 7.4 y 6.2 (procedimiento para encontrar el más grande de tres enteros), tanto para ver las diferencias entre una macro y un procedimiento, como para notar los efectos de las pseudoinstrucciones para hacer los programas más fáciles de escribir y entender.

Ejemplo 7.4: Macro para encontrar el más grande de tres valores Suponga que con frecuencia necesita determinar el más grande de tres valores en registros para poner el resultado en un cuarto registro. Escriba una macro `mx3r` para este propósito, siendo los parámetros el registro resultado y los tres registros operando.

Solución: La siguiente definición de macro usa sólo pseudoinstrucciones:

```
.macro mx3r(m,a1,a2,a3) # nombre de macro y argumento(s)
move    m,a1           # supone (a1) es el más grande;
bge     m,a2,+4         # si (a2) no es más grande, ignorarlo
move    m,a2           # sino fijar m = (a2)
bge     m,a3,+4         # si (a3) no es más grande, ignorarlo
move    m,a3           # sino fijar m = (a3)
.end_macro              # terminador de macro
```

Si la macro se usa como `mx3r($t0,$s0,$s4,$s3)`, el ensamblador sustituye los argumentos `m`, `a1`, `a2`, y `a3` en el texto de la macro con `$t0`, `$s0`, `$s4`, `$s3`, respectivamente, lo que produce las siguientes siete instrucciones en lugar de la macro (observe que las pseudoinstrucciones en la definición de macro se sustituyeron con instrucciones regulares):

```
addi    $t0,$s0,$zero   # supone ($s0) es más grande; $t0 = ($s0)
slt     $at,$t0,$s4     # ¿($t0) < ($s4)?
beq     $at,$zero,+4    # sino es, ignora ($s4)
addi    $t0,$s4,$zero   # sino fija $t0 = ($s4)
slt     $at,$t0,$s3     # ¿($t0) < ($s3)?
beq     $at,$zero,+4    # sino es, ignora ($s3)
addi    $t0,$s3,$zero   # sino fija $t0 = ($s3)
```

En otro lugar en el programa, la macro se usa como `mx3r($t5,$s2,$v0,$v1)`, lo que conduce a las mismas siete instrucciones que se insertan en el programa, pero con diferentes especificaciones de registro. Esto es similar a la llamada de procedimiento; por ende, los parámetros de macro a veces se llaman “parámetros formales”, como es común para subrutinas o procedimientos.

7.5 Ligado y cargado

Un programa, sea en lenguaje ensamblador o en lenguaje de alto nivel, consta de múltiples módulos que con frecuencia se diseñan para diferentes grupos en distintos momentos. Por ejemplo, una compañía de software que diseña un nuevo programa de aplicación puede reutilizar muchos procedimientos

previamente desarrollados como parte del nuevo programa. De manera similar, las *rutinas de librería* (por ejemplo, aquellas para cálculos de funciones matemáticas comunes o la realización de servicios útiles como entrada/salida) se incorporan en los programas al referirlos donde se necesiten. Para no reensamblar (o recompilar, para el caso de lenguajes de alto nivel) todos los módulos con cada pequeña modificación en uno de los componentes, se ofrece un mecanismo que permite construir, ensamblar y poner a prueba los módulos por separado. Justo antes de colocar estos módulos en la memoria para formar un *programa ejecutable*, las piezas se vinculan y se resuelven las referencias entre ellos, mediante un programa *vinculador* especial.

Cada uno de los módulos a vincular tendrá información en una sección de encabezado especial acerca del tamaño de los segmentos de texto (instrucciones) y de datos. También tendrá secciones que contengan información de reubicación y tabla de símbolos. Estas piezas de información adicionales no serán parte del programa ejecutable sino que se usan para permitir al vinculador realizar su tarea de manera correcta y eficiente. Entre otras funciones, el vinculador determina cuáles localidades de memoria ocupará cada uno de los módulos y ajustará (reubicará) todas las direcciones dentro del programa para corresponder a las localidades asignadas de los módulos. El programa vinculado combinado también tendrá tamaño y otra información relevante a usar por el *cargador*, que se describe brevemente.

Una de las funciones más importantes del vinculador es garantizar que las etiquetas en todos los módulos se interpreten (resuelvan) de manera adecuada. Si, por ejemplo, una instrucción `jal` especifica un blanco simbólico que no está definido en el módulo mismo, se busca en los otros módulos a vincular para determinar si alguno de ellos tiene un símbolo externo que se ajuste al nombre simbólico no resuelto. Si ninguno de los módulos resuelve la referencia indefinida, entonces se consultan las librerías de programa del sistema para ver si el programador pretendió el uso de una rutina de librería. Si algún símbolo permanece sin resolver al final de este proceso, la vinculación fracasa y se señala un error al usuario. La resolución exitosa de todas esas referencias por parte del vinculador se sigue por:

- Determinación del lugar de los segmentos de texto y datos en memoria.

- Evaluación de todas las direcciones de datos y etiquetas de instrucción.

- Formación de un programa ejecutable sin referencias no resueltas.

La salida del vinculador no va a la memoria principal donde se pueda ejecutar. Más bien, toma la forma de un archivo que se almacena en memoria secundaria. El *cargador* transfiere este archivo en la memoria principal para su ejecución. Puesto que el mismo programa se puede colocar en diferentes áreas de memoria durante cada ejecución, dependiendo de dónde hay espacio disponible, el cargador es responsable de ajustar todas las direcciones en el programa para que correspondan a la localidad real en memoria del programa. El vinculador supone que el programa se cargará comenzando con la localidad 0 en memoria. Si, en lugar de ello, la localidad de inicio es L , entonces todas las direcciones absolutas se deben correr por L . Este proceso, denominado *reubicación (relocation)*, involucra añadir una cantidad constante a todas las direcciones absolutas dentro de las instrucciones conforme se cargan en memoria. Asimismo, las direcciones relativas no cambian, de esta manera una bifurcación a 25 ubicaciones adelante, o 12 ubicaciones atrás, no cambia durante la reubicación.

Por tanto, el cargador es responsable de lo siguiente:

- Determinación de las necesidades de memoria del programa a partir de su encabezado.

- Copiar en memoria texto y datos del archivo de programa ejecutable.

- Modificar (correr) direcciones, donde se necesite, durante el copiado.

- Colocar parámetros de programa en la *pila* (como en un llamado de procedimiento).

- Inicializar todos los registros de máquina, incluso el apuntador de pila.

- Saltar a una rutina de arranque que llama la rutina principal del programa.

Observe que el programa se trata como un procedimiento que se llama desde una rutina del sistema operativo. Hasta la terminación, el programa regresa el control a la misma rutina, que entonces ejecuta un llamado de sistema *exit* (salida). Esto último permite el paso de datos y otros parámetros al programa a través del mismo mecanismo de pila usado por los llamados de procedimiento. Inicializar los registros de máquina significa limpiarlos a 0, excepto para el apuntador de *pila*, que se fija para apuntar a lo alto de la pila después de que todos los parámetros de programa se empujaron en ella.

■ 7.6 Corrida de programas ensamblador

En la práctica computacional moderna, los programas en lenguaje ensamblador o de máquina se generan mediante compiladores. Los programadores rara vez escriben directamente en lenguaje ensamblador porque éste es un proceso que consume tiempo y es proclive al error. No obstante, en ciertas situaciones prácticas (por ejemplo, para un módulo de programa que por rendimiento se debe optimizar) se puede programar en lenguaje ensamblador. Desde luego, también existe la razón pedagógica: para aprender lo suficiente acerca del lenguaje de máquina y ser capaz de entender y apreciar los conflictos en el diseño de computadoras, se debe escribir al menos algunos programas sencillos en lenguaje ensamblador. En este sentido, se introduce el simulador SPIM que pone a prueba los programas en lenguaje ensamblador MiniMIPS al correrlos para su desarrollo, así como para observar los resultados obtenidos y rastrear la ejecución en caso de problemas y facilitar de esa manera la depuración (*debugging*).

Los lectores que no tengan el propósito de diseñar y correr programas en lenguaje ensamblador MiniMIPS, pueden ignorar el resto de esta sección.

SPIM, que obtiene su nombre al invertir “MIPS”, representa un simulador para el lenguaje ensamblador MIPS R2000/R3000. El conjunto de instrucciones aceptado por SPIM es más grande que el subconjunto MiniMIPS cubierto en este libro, pero esto último no origina problemas para correr los programas. Esta situación se compara con la de alguien con un limitado vocabulario de inglés que habla con quien domina el idioma; este último entenderá lo que se habla, siempre que el pequeño subconjunto de palabras se use correctamente. Existen tres versiones de SPIM disponibles para su descarga gratuita:

PCSpim	para máquinas Windows
xspim	para Macintosh OS X
spim	para sistemas Unix o Linux

Usted puede descargar SPIM al visitar <http://www.cs.wisc.edu/~larus/spim.html> y seguir las instrucciones que ahí se dan. Lea la nota de derechos de autor y condiciones de uso que se encuentra al fondo de la citada página Web antes de comenzar a usar el programa.

La siguiente descripción se basa en PCSpim, que fue adaptada por David A. Carley de las otras dos versiones desarrolladas por el Dr. James R. Larus (anteriormente en la Universidad de Wisconsin, Madison, y ahora con Microsoft Research).

La figura 7.3 muestra la interfaz de usuario de PCSpim. Junto a las barras de menú y de herramientas en la parte superior, y la barra de estado en la parte inferior de la ventana de PCSpim, existen cuatro paneles en los que se muestra información acerca del programa ensamblador y su ejecución:

Los contenidos de registro se muestran en el panel superior.

El segmento de texto del programa se muestra en el segundo panel, cada línea contiene:

[dirección de memoria hexa]	contenido de instrucción hexa	<i>opcode</i> y parámetros
-----------------------------	-------------------------------	----------------------------

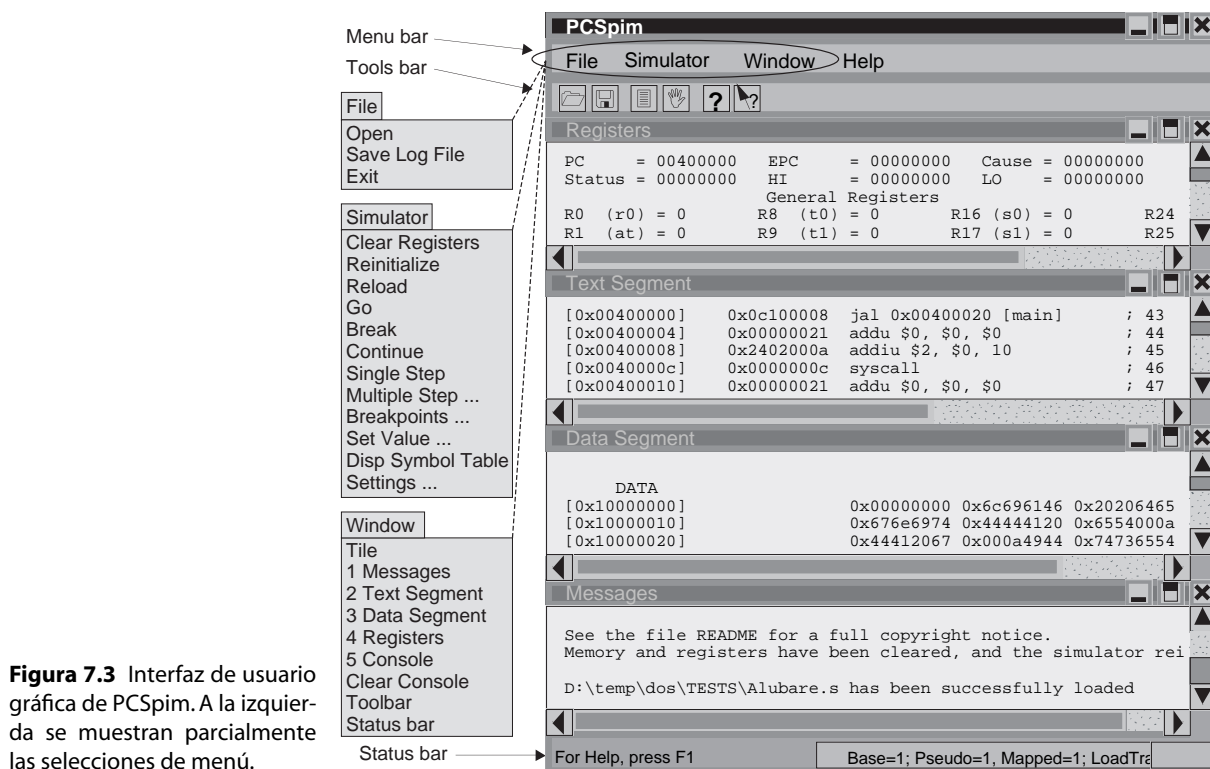


Figura 7.3 Interfaz de usuario gráfica de PCSpim. A la izquierda se muestran parcialmente las selecciones de menú.

El segmento de datos del programa se muestra en el tercer panel, y cada línea contiene una dirección de memoria hexa y los contenidos de cuatro palabras (16 bytes) en formato hexa.

Los mensajes producidos por SPIM se despliegan en el cuarto panel.

El simulador tiene ciertos ajustes predeterminados para sus características configurables. Éstos están bien aplicados para el objetivo del texto, así que no se discutirá cómo se modifican. El sustento para las modificaciones se descarga junto con el simulador.

Para cargar y correr en PCSpim un programa en lenguaje ensamblador MiniMIPS, primero debe componer el programa en un archivo de texto. Luego se puede abrir el archivo a través del menú *file* de PCSpim y esto hará que los segmentos de texto y de datos del programa aparezcan en los dos paneles de en medio de la ventana de PCSpim. Enseguida se puede ingresar a varias funciones de simulador, como *Go* para comenzar la ejecución del programa o *Break* para detenerlo, mediante el menú *Simulator* de PCSpim.

En la parte cuatro del libro se estudiarán con detalle entrada/salida e interrupciones. Aquí se proporciona un breve panorama de estos conceptos como para permitir la introducción de mecanismos PCSpim para entrada y salida de datos y los significados de las designaciones “con desbordamiento” y “sin desbordamiento” para algunas instrucciones de las tablas 5.1 y 6.2. Las instrucciones para entrada y salida de datos permitirán escribir programas completos que incluyan adquisición automática de los parámetros necesarios (por decir, entrada desde un teclado) y presentación de resultados calculados (o sea, salida a una pantalla de monitor).

Al igual que muchas computadoras modernas, MiniMIPS tiene I/O mapeado por memoria. Eso significa que ciertos buffers de datos y registros de estatus dentro de los dispositivos I/O, que se requieren para iniciar y controlar el proceso I/O, se visualizan como localidades de memoria. Por ende, si se co-

nocen las localidades específicas de memoria asignadas a un dispositivo de entrada como el teclado, se puede cuestionar acerca del estatus del dispositivo y transferir datos desde su buffer de datos hacia uno de los registros generales con el uso de la instrucción *load word* (cargar palabra). Sin embargo, este nivel de detalle en las operaciones I/O sólo es de interés en rutinas I/O conocidas como *manipuladores de dispositivo* (*device handlers*). En el capítulo 22 se estudiarán algunos aspectos de tales operaciones I/O. En el nivel de programación de lenguaje ensamblador, I/O con frecuencia se manipula a través de llamadas de sistema, ello significa que el programa solicita los servicios del sistema operativo para realizar la operación de entrada o salida deseada.

El modo I/O por defecto de PCSpim se realiza mediante una instrucción de llamada de sistema con el *opcode* simbólico *syscall*. La instrucción en lenguaje de máquina para *syscall* es de tipo R, con todos sus campos fijados en 0, excepto el campo *function*, que contiene 12. Una llamada de sistema se caracteriza por un código entero en [1, 10] que se coloca en el registro *\$v0* antes de la instrucción *syscall*. La tabla 7.2 contiene las funciones asociadas con estas diez llamadas de sistema.

Las operaciones entrada y salida hacen que PCSpim abra una nueva ventana llamada *Console*. Cualquier salida desde el programa aparece en la consola; todas las entradas al programa deben entrar del mismo modo en esta ventana. La llamada de sistema *allocate memory* (asignar memoria) (*syscall* con 9 en *\$v0*) resulta en un apuntador hacia un bloque de memoria que contiene *n* bytes adicionales, donde el número deseado *n* se proporciona en el registro *\$a0*. La llamada de sistema *exit from program* (salir de programa) (*syscall* con 10 en *\$v0*) hace que la ejecución del programa se termine.

Una operación aritmética, como la suma, puede producir un resultado que sea muy grande para ajustar en el registro destino especificado. Este evento se llama *overflow* (desbordamiento) y causa que se produzca una suma inválida. Obviamente, si se continúa el cálculo con esta última, existirá una gran probabilidad de producir resultados sin sentido del programa. Para evitar esto, el hardware de MiniMIPS reconoce que existe un desbordamiento y llama una rutina especial del sistema operativo conocida como manipulador de interrupción. Esta interrupción del flujo normal del programa y la consecuente transferencia de control al sistema operativo se refiere como una *interrupción* o *excepción*. El manipulador de interrupción puede seguir varios cursos de acción, según la naturaleza o causa de la interrupción. Esto último se discutirá en el capítulo 24. Hasta entonces, es suficiente saber que MiniMIPS asume que el manipulador de interrupción comienza en la dirección 0x80000080 en memoria y le transferirá el control si ocurriera una interrupción. La transferencia de control a través de un saldo incondicional (muy parecido a *jal*) y la dirección de retorno, o el valor en el contador de programa

■ **TABLA 7.2** Funciones entrada/salida y control de *syscall* en PCSpim.

Clase	(<i>\$v0</i>)	Función	Argumento(s)	Resultado
Salida	1	Imprimir entero	Entero en <i>\$a0</i>	Despliega entero
	2	Imprimir punto flotante	Flotante en <i>\$f12</i>	Despliega flotante
	3	Imprimir doble flotante	Doble flotante en <i>\$f12</i> , <i>\$f13</i>	Despliega doble flotante
	4	Imprimir cadena	Apuntador en <i>\$a0</i>	Despliega cadena terminada en nulo
Entrada	5	Leer entero		Regresa entero en <i>\$v0</i>
	6	Leer punto flotante		Regresa flotante en <i>\$f0</i>
	7	Leer doble flotante		Regresa doble flotante en <i>\$f0</i> , <i>\$f1</i>
	8	Leer cadena	Apuntador en <i>\$a0</i> , longitud en <i>\$a1</i>	Regresa cadena en el buffer del apuntador
Control	9	Alojar memoria	Número de bytes en <i>\$a0</i>	Apuntador a bloque de memoria en <i>\$v0</i>
	10	Salir de programa		Termina ejecución de programa

al momento de la interrupción, se salva automáticamente en un registro especial llamado *exception program counter* (contador de excepción de programa) o EPC en el coprocesador 0 (figura 5.1).

A algunas instrucciones se les da la designación “sin desbordamiento” porque usualmente se utilizan cuando el desbordamiento es imposible o indeseable. Por ejemplo, uno de los usos principales de la aritmética con números sin signo es el cálculo de dirección. Agregar una dirección base a un *offset* en bytes (cuatro veces el *offset* en palabras) puede producir una dirección en memoria correspondiente a un nuevo elemento de arreglo. En condiciones normales, esta dirección calculada no superará el límite del espacio de dirección de la computadora. Si lo hace, el hardware incluye otros mecanismos para detectar una dirección inválida y no es necesaria una indicación de desbordamiento de la unidad de ejecución de instrucciones.

Desde el enfoque de la escritura de programas en lenguaje ensamblador, se puede ignorar la posibilidad del desbordamiento, y la interrupción asociada, siempre que se tenga cuidado de que los resultados y valores intermedios no sean demasiado largos en magnitud. Esto es fácil de hacer para cálculos simples y para los programas que se trataron aquí.

PROBLEMAS

7.1 Directivas de ensamblador

Escriba directivas de ensamblador para lograr cada uno de los objetivos siguientes:

- a) Establecimiento de cuatro mensajes de error, cada uno de los cuales tiene exactamente 32 caracteres de largo, de modo que el programa puede imprimir el *i*-ésimo mensaje de error ($0 \leq i \leq 3$) con base en un índice numérico en un registro. *Sugerencia:* El índice en el registro puede ser corrido a la izquierda por cinco bits antes de usarse como *offset*.
- b) Establecimiento de las constantes enteras *least* y *most*, con valores 25 y 570, contra las que se puede verificar la validez de los datos de entrada dentro del programa.
- c) Apartar espacio para una cadena de caracteres de longitud que no supere 256 símbolos, incluidos un terminador nulo, para el caso de que exista.
- d) Apartar espacio para una matriz entera de 20×20 , con el fin de almacenar en memoria en un orden de mayor renglón.

- e) Apartar espacio para una imagen con un millón de píxeles, cada una se especifica mediante un código de color de ocho bits y otro código de brillantez de ocho bits.

7.2 Definición de seudoinstrucciones

De las 20 seudoinstrucciones de la tabla 7.1, dos (*not*, *abs*) se discutieron al comienzo de la sección 7.3 y cuatro más (*neg*, *rem*, *li*, *blt*) se definieron en el ejemplo 7.2. Proporcione instrucciones o secuencias de instrucciones MiniMIPS equivalentes para las restantes 14 seudoinstrucciones; lo anterior constituye las partes a-n del problema, en orden de aparición en la tabla 7.1.

7.3 Seudoinstrucciones adicionales

Las siguientes son algunas seudoinstrucciones adicionales que uno podría definir para MiniMIPS. En cada caso, proporcione una instrucción o secuencia de instrucciones MiniMIPS equivalente con el efecto deseado. *part*, *mulacc* es abreviatura de “multiplicar-acumular”.

<code>parta: beqz reg,L</code>	<code># si (reg)=0, ir a L</code>
<code>partb: bnez reg,L</code>	<code># si (reg)≠0, ir a L</code>
<code>partc: bgtz reg,L</code>	<code># si (reg)>0, ir a L</code>
<code>partd: blez reg,L</code>	<code># si (reg)≤0, ir a L</code>
<code>parte: bgez reg,L</code>	<code># si (reg)≥0, ir a L</code>
<code>partf: double regd,regs</code>	<code># regd = 2×(regs)</code>
<code>partg: triple regd,regs</code>	<code># regd = 3×(regs)</code>
<code>parth: mulacc regd,reg1,reg2</code>	<code># regd = (regd) + (reg1)×(reg2)</code>

7.4 Seudoinstrucciones complejas

Las pseudoinstrucciones de los problemas 7.2 y 7.3 recuerdan instrucciones de ensamblado MiniMIPS regulares. En este problema se propone un número de pseudoinstrucciones que realizan funciones más complejas que no recuerdan instrucciones ordinarias y podrían no implementarse como tales dentro de las restricciones de formato de las instrucciones MiniMIPS. Las siguientes pseudoinstrucciones realizan las funciones “incremento de palabra de memoria”, “*fetch* y suma” y “*fetch* y suma inmediata”. En cada caso, proporcione una secuencia de instrucciones MiniMIPS equivalente con el efecto deseado.

```
parta: incmem reg,imm      # mem[(reg)+imm] = mem[(reg)+imm] + 1
partb: ftcha  reg1,reg2,imm # mem[(reg1)+imm]=mem[(reg1)+imm]+(reg2)
partc: ftchai reg,imm1,imm2 # mem[(reg)+imm1]=mem[(reg)+imm1]+imm2
```

7.5 Entrada y salida

- Convierta el programa de clasificación de selección del ejemplo 7.3 en un programa que reciba la lista de enteros a clasificar como entrada y que despliegue su salida.
- Corra el programa de la parte a) en el simulador SPIM y use una lista de entrada de al menos 20 números.

7.6 Nuevas pseudoinstrucciones

Proponga al menos dos pseudoinstrucciones útiles para MiniMIPS que no aparezcan en la tabla 7.1. ¿Por qué cree que sus pseudoinstrucciones son útiles? ¿Cómo se convierten sus pseudoinstrucciones a instrucciones o secuencias de instrucciones MiniMIPS equivalentes?

7.7 Cómo encontrar el n -ésimo número de Fibonacci

Escriba un programa MiniMIPS completo que acepte n como entrada y produzca el n -ésimo número de Fibonacci como salida. Los números de Fibonacci se definen recursivamente mediante la fórmula $F_n = F_{n-1} + F_{n-2}$, con $F_0 = 0$ y $F_1 = 1$.

- Escriba su programa con un procedimiento recursivo.
- Escriba su programa sin recursividad.
- Compare los programas o las partes a) y b) y discuta.

7.8 Programa de raíz cuadrada

Escriba un programa MiniMIPS completo que acepte un entero x como entrada y genere $\lfloor x^{1/2} \rfloor$ como sali-

da. El programa debe imprimir un mensaje adecuado si la entrada es negativa. Puesto que la raíz cuadrada de un número binario de 31 bits no es mayor que 46340, una estrategia de búsqueda binaria en el intervalo [0, 65, 536] puede llevar a la identificación de la raíz cuadrada en 16 o menos iteraciones. En búsqueda binaria, se determina el punto medio $(a + b)/2$ del intervalo de búsqueda $[a, b]$ y la búsqueda se restringe a $[a, (a + b)/2]$ o $[(a + b)/2, b]$, dependiendo del resultado de una comparación.

7.9 Cómo encontrar *máx* y *mín*

Escriba un programa MiniMIPS completo que acepte una secuencia de enteros en entrada y, después de la recepción de cada nuevo valor de entrada, despliegue los enteros más grande y más pequeño hasta el momento. Una entrada de 0 indica el final de los valores de entrada y no es un valor de entrada en sí mismo. Observe que no necesita conservar todos los enteros en memoria.

7.10 Tabla de símbolos

- Muestre la tabla de símbolos elaborada por el ensamblador MiniMIPS para el programa del ejemplo 7.3. Ignore las partes de entrada y salida.
- Repita la parte a), esta vez suponiendo que se han incluido operaciones adecuadas de entrada y salida, de acuerdo con el problema 7.5.

7.11 Programa de escritura de cheques

Los cheques de una compañía se deben expedir mediante la escritura de la cantidad con palabras sustituidas por dígitos decimales (por ejemplo, “cinco dos siete dólares y cinco cero centavos”) para hacer más difícil la falsificación. Escriba un programa MiniMIPS completo para leer un entero no negativo, que represente una cantidad en centavos, como entrada y produzca la cantidad equivalente en palabras como salida, como sugiere el ejemplo de \$527.50 correspondiente a la entrada entera 52750.

7.12 Formación de triángulo

Escriba un programa MiniMIPS completo para leer tres enteros no negativos, presentados en orden arbitrario, como entradas y determine si pueden formar los lados de:

- a) Un triángulo
- b) Un triángulo recto
- c) Un triángulo agudo
- d) Un triángulo obtuso

7.13 Secuencias de enteros

Escriba un programa MiniMIPS completo para leer tres enteros no negativos, presentados en orden arbitrario, como entradas y determine si pueden formar términos consecutivos de:

- a) Una serie de Fibonacci (en la que cada término es la suma de los dos términos precedentes)
- b) Una progresión aritmética
- c) Una progresión geométrica

7.14 Factorización y prueba de primalidad

Escriba un programa MiniMIPS completo para aceptar un entero no negativo x en el rango $[0, 1\,000\,000]$ como entrada y despliegue sus factores, del más pequeño al más grande. Por ejemplo, la salida del programa para $x = 89$ debe ser “89” (que indica que x es un número primo) y su salida para $x = 126$ debe ser “2, 3, 3, 7”.

7.15 Conversión decimal a hexadecimal

Escriba un programa MiniMIPS completo para leer un entero sin signo con hasta seis dígitos decimales de la entrada y represéntelo en forma hexadecimal en la salida. Observe que, como consecuencia de que el entero de entrada encaja en una sola palabra de máquina, la conversión decimal a binario sucede como parte del proceso de entrada. Todo lo que necesita hacer es derivar la representación hexadecimal de la palabra binaria.

7.16 Cifrado de sustitución

Un cifrado de sustitución cambia un mensaje en texto plano a un texto cifrado mediante la sustitución de cada una de las 26 letras del alfabeto inglés por otra letra de acuerdo con una tabla de 26 entradas. Por simplicidad, se supone que el texto plano o el texto cifrado no contienen espacio, numerales, signos de puntuación o símbolos especiales. Escriba un programa MiniMIPS completo para leer una cadena de 26 símbolos que defina la tabla de sustitución (el primer símbolo es la sustitución de a, el segundo de b, etc.) y luego solicita repetidamente entradas en texto plano, ello propicia el texto cifrado equivalente como salida. Cada entrada de texto plano termina con un punto. La entrada de cualquier símbolo distinto a una letra minúscula o punto termina la ejecución del programa.

REFERENCIAS Y LECTURAS SUGERIDAS

- | | |
|----------|--|
| [Henn03] | Hennessy, J. L. y D. A. Patterson, <i>Computer Architecture: A Quantitative Approach</i> , Morgan Kaufmann, 3a. ed., 2003. |
| [Kane92] | Kane, G. y J. Heinrich, <i>MIPS RISC Architecture</i> , Prentice Hall, 1992. |
| [MIPS] | Sitio Web de tecnologías MIPS. Siga las ligas de arquitectura y documentación en http://www.mips.com/ |
| [Pat98] | Patterson, D. A. y J. L. Hennessy, <i>Computer Organization and Design: The Hardware/Software Interface</i> , Morgan Kaufmann, 2a. ed., 1998. |
| [Sail96] | Sailer, P. M. y D. R. Kaeli, <i>The DLX Instruction Set Architecture Handbook</i> , Morgan Kaufmann, 1996. |
| [Silb02] | Silberschatz, A. P. B. Galvin, and G. Gagne, <i>Operating System Concepts</i> , Wiley, 6a. ed., 2002. |
| [SPIM] | SPIM, simulador de software de descarga gratuita para un subconjunto del conjunto de instrucciones MIPS; más información en la sección 7.6.
http://www.cs.wisc.edu/~larus/spim.html |

VARIACIONES EN EL CONJUNTO DE INSTRUCCIONES

“Los ingenieros que fueron pioneros en el movimiento RISC tienen una ventaja sobre sus predecesores, quienes diseñaron los primeros conjuntos de instrucciones. En el intervalo, las técnicas de simulación maduraron, y la potencia de cómputo necesaria para su explotación resultó fácilmente disponible. Fue posible valorar la eficiencia de un diseño sin realmente implementarlo... Pronto se volvió aparente que la intuición sin auxilio era una guía muy mala.”

Maurice Wilkes, Computing Perspectives

“Todo se debe hacer tan simple como sea posible, pero no más simple.”

Albert Einstein

TEMAS DEL CAPÍTULO

- 8.1 Instrucciones complejas
- 8.2 Modos de direccionamiento alterno
- 8.3 Variaciones en formatos de instrucción
- 8.4 Diseño y evolución del conjunto de instrucciones
- 8.5 La dicotomía RISC/CISC
- 8.6 Dónde dibujar la línea

El conjunto de instrucciones MiniMIPS, aunque bastante típico entre los que usan muchos procesadores modernos, sólo es un ejemplo. En este capítulo se discuten las formas en que la arquitectura del conjunto de instrucciones puede ser diferente de los MiniMIPS, con el propósito de llegar a un cuadro equilibrado y completo del estado de la práctica en el diseño del conjunto de instrucciones. Entre otras cosas, se observan modos de direccionamiento más elaborados y variaciones en los formatos de instrucciones. Luego se examinan características deseables de un conjunto de instrucciones y se analiza lo idóneo de las filosofías RISC y CISC a este respecto. Se concluye con la descripción de una interesante computadora de una sola instrucción, lo más avanzado en arquitectura RISC.

■ 8.1 Instrucciones complejas

Las instrucciones MiniMIPS que se presentan en los capítulos 5 a 7 realizan tareas simples como una operación aritmética/lógica sencilla, copiar elementos de datos básicos o transferir el control. Esto es cierto en todas las computadoras modernas. Conceptualmente no hay nada que impida definir una instrucción que realice una tarea compleja como:

```

chksum regd,reg1,reg2  # fija regd en checksum (i.e., XOR)
                        # de todos los bytes en un arreglo cuya
                        # dirección
                        # de inicio (fin) está en reg1 (reg2)
sortup reg1,reg2       # ordena las palabras enteras en arreglos
                        # cuya dirección de inicio (fin) está en
                        # reg1 (reg2) en orden no descendiente

```

Sin embargo, cualquiera de tales instrucciones complejas se descompondrá a pasos más simples para la ejecución de hardware. Por ejemplo, la instrucción `chksum` es probable que se ejecute al realizar lo equivalente de un ciclo de software en hardware, esto es XOR izar el siguiente byte a una suma de verificación (*checksum*) que corre, incrementar un apuntador, comparar el apuntador contra la dirección final y repetir si no son iguales. La cuestión, entonces, es si vale la pena definir tales instrucciones complejas.

En el aspecto positivo, el uso de instrucciones más complejas conduce a programas con textos más cortos y menores requerimientos de memoria. En las primeras computadoras, las memorias (tanto primarias como secundarias) eran bastante pequeñas, esto último ofreció un fuerte incentivo para usar instrucciones complejas. Además, realizar un cálculo repetitivo en hardware probablemente es más rápido que su contraparte en software, debido al tiempo que se ahorra como resultado de *fetching* y decodificar una instrucción en lugar de muchas.

En el aspecto negativo, dominar una diversidad de instrucciones complejas y usarlas en lugares adecuados, tanto en programación manual como en compilación, es un gran desafío. Además, como se discute extensamente en la sección 8.5, la inclusión de instrucciones complejas en un conjunto de instrucciones podría hacer que incluso la instrucción más simple tome más tiempo de ejecución, y nulifique parte del potencial ganado. Finalmente, una instrucción como `sortup` es probable que tenga utilidad limitada, pues el algoritmo de clasificación implícito en su implementación en hardware paso a paso puede no ser la mejor en todas las situaciones posibles.

Por tanto, los arquitectos de computadoras tienen mucho cuidado al elegir cuáles instrucciones complejas se deben incluir en un conjunto de instrucción para garantizar que los beneficios sean mayores que los inconvenientes. La tabla 8.1 menciona algunas de las instrucciones complejas que existen en algunos procesadores populares.

La complejidad de una instrucción es, desde luego, una noción relativa. Lo que es complejo en un contexto de aplicación, o con una tecnología de implementación, puede ser más bien simple en otro escenario. También existen diferentes aspectos de complejidad. Una instrucción puede ser muy compleja para describir y comprender (y, por tanto, difícil de usar adecuadamente), pero bastante fácil de decodificar y ejecutar. La instrucción `CS` del IBM System/370 en la tabla 8.1 es una de éstas. Su inclusión en el conjunto de instrucciones fue motivada por la necesidad de manejar una cola compartida en un ambiente de multiprocesamiento [Giff87]. Se ejecuta fácilmente como la concatenación de tres operaciones simples: cargar en un buffer, comparar y copiar o almacenar, dependiendo del resultado de la comparación. En este contexto, una instrucción que parezca simple y fácil de comprender o usar, puede requerir pasos complejos y mucho consumo de tiempo para su ejecución. Las dos instrucciones que se introducen al comienzo de esta sección ofrecen buenos ejemplos.

Aunque no hay acuerdo acerca de la definición estándar para “instrucción compleja” en la literatura de arquitectura de computadoras, la siguiente refleja más o menos la esencia de lo que la mayoría de las personas entienden por el término: una instrucción compleja es aquella que potencialmente puede realizar múltiples accesos de memoria durante su ejecución. En virtud de que cada acceso de memoria necesita cálculo o actualización de dirección, además de la operación de lectura o escritura de memoria física, la ejecución de tales instrucciones puede ser más lenta que las instrucciones MiniMIPS de la tabla 6.2. Sin embargo, la complejidad se puede presentar sin acceso alguno a la memoria en absoluto. La instrucción

■ **TABLA 8.1** Ejemplos de instrucciones complejas en dos populares microprocesadores modernos (Pentium, PowerPC) y dos familias de computadoras de significado histórico (System/360-370, VAX).

Máquina	Instrucción	Efecto
Intel Pentium	MOVS	Mueve un elemento en una cadena de bytes, palabras o dobles palabras con el uso de las direcciones especificadas en dos apuntadores de registro; después de la operación, incrementa o disminuye los registros para apuntar al siguiente elemento de la cadena.
	BOUND	Verifica que el operando 1 esté dentro de los límites inferior y superior almacenados en dos palabras de memoria consecutivas referenciadas por el operando 2 (útil para verificación de índice de arreglo); si el operando 1 está fuera de las cotas, ocurre una interrupción.
	INVD, WBINVD	Limpia la memoria de caché interna al invalidar todas las entradas (para WBINVD, primero escribe todas las líneas de caché sucia a memoria). Estas instrucciones tendrán más sentido después de estudiar las memorias de caché en el capítulo 18.
IBM/Motorola Power PC	lmw	Carga palabras múltiples en localidades de memoria consecutivas en registros, desde el registro destino especificado hasta el registro 31 de propósito general.
	lswx	Carga una cadena de bytes en registros, cuatro bytes por registro, comenzando con el registro destino especificado; da vuelta del registro 31 al registro 0.
	cntlzd	Cuenta el número de 0 consecutivos en un registro fuente específico, comenzando con la posición de bit 0 y coloca la cuenta en un registro de destino.
	rldic	Rota hacia la izquierda un registro de doble palabra específico, opera AND el resultado con una máscara específica y almacena en un registro de destino.
IBM System/360-370	MVZ	Mueve zona: copia un número específico de bytes de una región de memoria a otra; cada una de ambas regiones se define mediante una dirección de comienzo compuesta de un registro base y un <i>offset</i> .
	TS	Prueba y establece: lee un bit de control de memoria y fija el bit a 1 si es 0, todo en una acción atómica ininterrumpible (se usa para acceso de control a un recurso compartido, donde 0/1 significa que el recurso está disponible/cerrado).
	CS	Compara e intercambia: compara el contenido de un registro con el de una ubicación de memoria; si es distinto, carga la palabra de memoria en el registro, de otro modo almacena el contenido de un registro diferente en la misma localidad de memoria.
Digital VAX	MOVTC	Mueve caracteres traducidos: copia una cadena de caracteres de un área de memoria a otra, y traduce cada carácter mediante una tabla de consulta (<i>lookup</i>) cuya localidad se especifica como parte de la instrucción.
	POLYD	Evaluación polinomial con doble aritmética de punto flotante: evalúa un polinomio en x , con muy alta precisión en resultados intermedios, con el uso de una tabla de coeficientes cuya localidad en memoria está dada dentro de la instrucción.
	ADDP6	Suma cadenas decimales empaquetadas (enteros base 10, con dos dígitos BCD almacenados por byte); cada uno de las dos cadenas decimales fuente y la cadena de destino se especifican mediante su longitud y ubicación, para un total de seis operandos.

cntlzd de la IBM/Motorola PowerPC en la tabla 8.1 proporciona un ejemplo excelente. Esta instrucción es compleja si el conteo de bits se realiza mediante múltiples ciclos de corrimiento condicional. Sin embargo, no es compleja si un circuito especial de conteo de ceros iniciales se implementa dentro de la ALU.

■ 8.2 Modos de direccionamiento alternativo

Los modos de direccionamiento de los MiniMIPS están limitadas de dos formas. Primero, sólo se usan seis modos: implícito, inmediato, de registro, base (+ *offset*), relativo a PC y directo (vea la figura 5.11). Segundo, incluso estos modos de direccionamiento limitados no se pueden usar en todas partes; por ejemplo, las instrucciones aritméticas sólo permiten los modos inmediato y de registro. Estas limitaciones se encuentran en muchas arquitecturas tipo RISC, que tienen la intención de acelerar las combinaciones de operaciones y direccionamiento de operando más comúnmente usadas (sección 8.5).

Las competidoras arquitecturas tipo CISC permiten usar los modos anteriores más ampliamente y también pueden tener modos de direccionamiento adicionales. Primero se revisa cómo se pueden usar

los seis modos de direccionamiento de MiniMIPS y luego se enfocará el análisis en dos importantes modos de direccionamiento que se pierden en los MiniMIPS: indizado e indirecto.

El **direccionamiento implícito** es mucho más versátil de lo que se pueda pensar. Han existido computadoras que usaron este modo exclusivamente para todas las operaciones aritméticas. En las *máquinas de pila*, todos los operandos requeridos de una instrucción aritmético/lógica se toman de lo alto de la pila y cualquier resultado también se coloca ahí. Una instrucción de este tipo sólo necesita un *opcode* y a veces se le conoce como *instrucción de dirección 0*. Desde luego, se necesitan otras formas de direccionamiento para tener la capacidad de mover datos en la memoria; por ejemplo, desde una localidad arbitraria a lo alto de la pila y viceversa (equivalentes a las instrucciones *load* y *store* en MiniMIPS). Sin embargo, todavía se ahorra gran cantidad de espacio en las instrucciones que normalmente requerirían especificaciones para operandos y localidades de resultado. Otros ejemplos de direccionamiento implícito se encuentran en las instrucciones *increment* (incremento) y *decrement* (disminución) de algunas máquinas que son los equivalentes del *addi* de MiniMIPS, con el operando inmediato implícitamente especificado como $+1$ o -1 .

El **direccionamiento inmediato** se usa para un operando constante que es muy pequeño como para encajar en la instrucción misma. Sin direccionamiento inmediato, tales constantes tendrían que almacenarse en registros (por tanto, dejan menos registros para otros usos) o recuperar de la memoria (por tanto, requieren más bits de direccionamiento o mayor latencia). El costo de tener direccionamiento inmediato es que habrán múltiples formas de cada instrucción aritmético/lógica, ello tiende a complicar la decodificación de instrucciones y los circuitos de control. En general, los MiniMIPS sólo permiten operandos inmediatos de 16 bits, aunque, para instrucciones de corrimiento, se usa un operando inmediato de cinco bits, que designa la cantidad de corrimiento. Otros conjuntos de instrucciones, más complejos, pueden permitir operandos inmediatos de tamaños variables, desde bytes hasta dobles palabras. Sin embargo, esto complica aún más la decodificación de instrucciones y el control. Observe que si trata una instrucción como *addi rd, rs, 8* como un entero de 32 bits y se le suma 4, su operando inmediato cambiará a 12. Los *programas automodificables* de este tipo, bastante común en los primeros días de la computación digital, ya no se usan más porque complican enormemente la comprensión del usuario y la depuración de programas.

El **direccionamiento por registro** es el modo de direccionamiento más común porque se puede acceder más rápido a los operandos de registro y requiere menos bits para especificar, que los operandos de memoria. Una operación que requiere dos operandos y produce un resultado, todo en registros distintos, pueden encajar fácilmente en 32 bits (acaso incluso en 16 bits, para un número más pequeño de registros). Sin embargo, el registro de resultado no tiene que ser distinto de los registros de operando, pues lleva a un mayor ahorro en el número de bits necesarios. Por ejemplo, la operación $a = a + b$ sólo requiere dos especificaciones de registro. Las primeras máquinas tenían un registro especial conocido como *el acumulador*, que siempre proporcionaba un operando y recibía el resultado de cualquier operación aritmético/lógica. Las instrucciones aritméticas/lógicas en tales máquinas sólo requerían una especificación de registro. Note que el uso de un acumulador, o un registro que duplica tanto una fuente de operando como el destino del resultado, es una forma de direccionamiento implícito.

El **direccionamiento base**, también conocido como direccionamiento “base más *offset*”, puede tomar formas diferentes. En MiniMIPS, el valor base de un registro se agrega a un *offset* especificado como un operando inmediato, ello produce una dirección de memoria de 32 bits. Observe que llamar a un valor “base” y al otro *offset* es cuestión de convención; invertir los papeles de los dos valores no sólo es posible sino útil en la práctica. Objetivamente, la base es una constante, mientras que el *offset* es variable. De este modo, si el registro apunta al comienzo de un arreglo o lo alto de una *stack* mientras que el valor inmediato especifica a cuál elemento de arreglo o de *stack* se debe acceder, entonces el registro retiene la base y el valor inmediato es el *offset*. Lo inverso también es posible, siempre que la base se

pueda especificar como un operando inmediato. La última opción proporciona una forma limitada de *direccionamiento índice* (que se considerará dentro de poco), donde el registro actúa como un *registro índice*. Cuando la parte inmediata es 0, la dirección calculada es la misma que el valor en el registro especificado. A este tipo de direccionamiento a veces se le denomina *indirecto registro*, que implica que a la memoria se accede indirectamente a través de un registro, lo que es opuesto a directamente mediante el suministro de una dirección de memoria. De nuevo, las restricciones MiniMIPS de tener sólo operandos inmediatos de 16 bits se relaja en algunas arquitecturas de conjunto de instrucciones, que conduce a direccionamientos base más versátiles. En particular, con un operando inmediato de 32 bits, el direccionamiento base y el direccionamiento índice se pueden volver indistinguibles.

El **direccionamiento relativo** se usa cuando la dirección se especifica en relación con una dirección de referencias acordada de antemano. La única diferencia con el direccionamiento base es que la dirección de referencia es implícita en lugar de explícita. La forma de direccionamiento relativo más comúnmente usada toma el contenido del contador de programa (correspondiente a la instrucción actual o, con más frecuencia, a la instrucción siguiente) como el punto de referencia y se conoce como *direccionamiento relativo al PC*. Las máquinas más modernas usan direccionamiento relativo al PC para bifurcaciones, porque las direcciones destino de bifurcación usualmente están muy cerca del punto de bifurcación, ello especifica el destino con un pequeño *offset* con signo (*offset* negativo para bifurcaciones hacia atrás y *offset* positivo para bifurcaciones hacia adelante). En las máquinas que tienen instrucciones especiales para manipular la pila, el apuntador de *pila* se puede usar como punto de referencia. En algunas arquitecturas, el contador de programa forma parte del archivo de registro general; esto hace al direccionamiento relativo al PC muy parecido al direccionamiento base.

El **direccionamiento directo** constituye la forma más básica de direccionamiento. El tamaño de las primeras memorias de computadora se medía en miles de localidades, lo anterior significaba que incluso una instrucción de 16 bits tenía suficiente espacio para albergar una dirección de memoria completa; nadie sospechaba que una instrucción de 32 bits pudiera algún día ser inadecuada para el direccionamiento directo. Desde luego, se podría resolver el problema para el futuro previsible al hacer todas las instrucciones de 64 bits de ancho, pero esto sería un desperdicio para las instrucciones que no requieren una dirección de memoria directa. Las instrucciones de ancho variable que se usan en algunas máquinas ofrecen una solución, como lo hacen algunos de los modos de direccionamiento ya discutidos. MiniMIPS usa un esquema que permite lograr direccionamiento directo con direcciones de 32 bits en instrucciones de 32 bits. Usa dos bits implícitos de 0 en el extremo inferior, junto con cuatro bits del contador de programa en el extremo superior para convertir el campo *address* de 26 bits en una dirección de 32 bits completos. Los esquemas que se acercan al direccionamiento directo se conocen como *direccionamientoseudodirecto*.

El **direccionamiento indexado** toma su nombre de su principal aplicación para acceder a los elementos de un arreglo. Algunas de las primeras computadoras tenían uno o más registros que se dedicaban al indexado. Si el modo de direccionamiento era indexado, el contenido de uno de estos registros se agregaría a la dirección base, que se especificó directamente o a través de uno de los modos de direccionamiento anteriores. En algunos modelos posteriores, cualquier registro en el archivo de registro general se podría especificar como un registro índice. Como ya se mencionó, el direccionamiento base en MiniMIPS ofrece una forma limitada de indexar en la que la dirección base se especifica en 16 bits. El direccionamiento indexado convencional requiere que la base sea una dirección completa. Usualmente, la dirección completa que contiene la base está dada por especificación directa o se retiene en un registro. Para el último caso, la dirección efectiva se obtiene como la suma de los contenidos de dos registros. En virtud de que es común acceder al elemento siguiente de un arreglo después del actual, algunas máquinas ofrecen una forma más elaborada de direccionamiento indexado, ésta se denomina *direccionamiento actualizado*.

El **direccionamiento actualizado** constituye cualquier esquema de direccionamiento en el que uno de los registros involucrados en la formación de la dirección (por lo general un registro que se usa como índice)

ce) se actualiza de modo que el cálculo de una nueva dirección genera la dirección del siguiente elemento del arreglo. El direccionamiento actualizado requiere la suma de una constante (± 1 o \pm una potencia de 2) al registro designado. Esta capacidad ahorra una instrucción en muchos ciclos comunes y también puede acelerar la ejecución si la actualización del índice se realiza en paralelo con la ejecución de la parte principal de la instrucción. El direccionamiento actualizado se puede usar con el direccionamiento base, siempre que el componente inmediato se estime como la base y el componente de registro como el *offset*.

El **direccionamiento indirecto** requiere un proceso de dos etapas para obtener un operando. En la primera, se consulta la localidad designada (un registro o una palabra de memoria); ello produce una dirección en lugar de un operando. En la segunda, esta dirección se usa para recuperar el operando mismo. Si la primera etapa especifica un registro, el modo de direccionamiento se conoce como *indirecto registro*, que representa un caso especial del direccionamiento base con un *offset* de 0. Uno de los usos prominentes de direccionamiento indirecto es la construcción de *tablas de salto*. Una tabla de salto es un arreglo cuyos elementos son direcciones. Suponga que una de n secuencias de acciones se debe realizar dependiendo del valor de una variable entera j (esto es lo que hace la sentencia *case* en algunos lenguajes de programación de alto nivel). Luego, al colocar las direcciones de inicio de n secuencias de instrucción en un arreglo de longitud n y ejecutar un salto indirecto al j -ésimo elemento del arreglo, logra la tarea deseada. Al igual que los otros modos de direccionamiento complejos, el efecto del direccionamiento indirecto se puede lograr mediante modos de direccionamiento más simples.

La figura 8.1 bosqueja los nuevos modos de direccionamiento apenas discutidos (indexado, actualizado e indirecto) en un formato similar al de la figura 5.11 (que muestra los modos de direccionamiento implícito, inmediato, por registro, base, relativo al PC yseudodirecto). Observe que, aun cuando las instrucciones que se muestran en la figura 8.1 hacen pensar en instrucciones MiniMIPS, estos modos de direccionamiento no los soporta MiniMIPS.

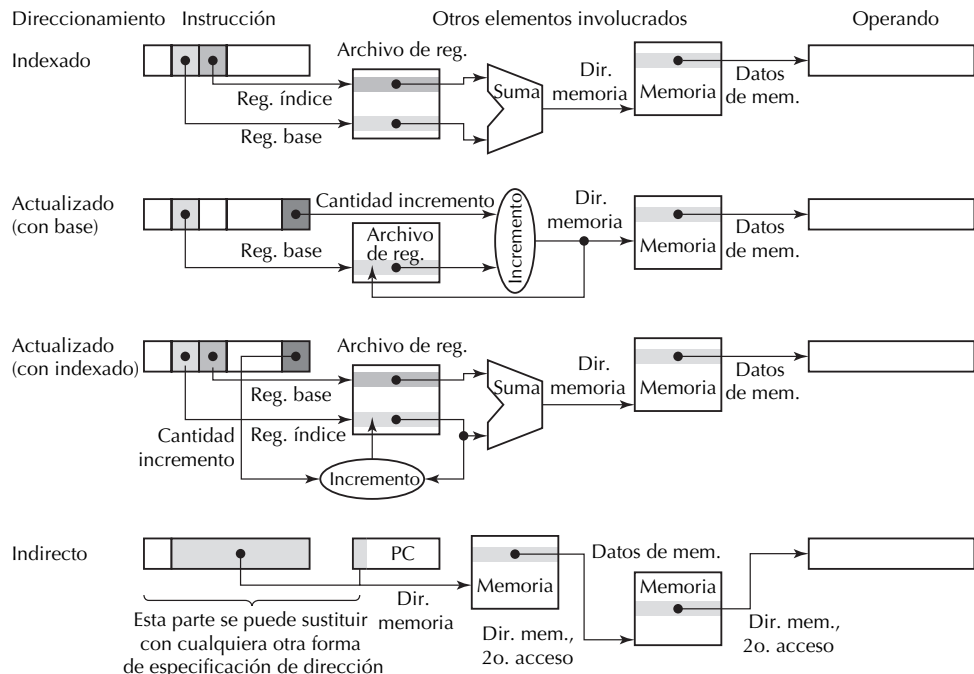


Figura 8.1 Representación esquemática de modos de direccionamiento más elaborados no soportados en MiniMIPS.

Al igual que en MiniMIPS, la mayoría de las máquinas ponen ciertas restricciones en los tipos de modo de direccionamiento que se pueden usar con cada instrucción o clase de instrucciones. La arquitectura VAX, que se introdujo a mediados de la década de 1970, incluyó la generalidad en el direccionamiento, en el sentido de que cada operando especificado en una instrucción podía utilizar cualesquiera modos de direccionamiento disponibles. Esto último condujo a un formato de instrucción variable, en especial porque, para algunas instrucciones, se podían especificar hasta seis operandos (tabla 8.1). Los formatos de instrucción se discuten en la siguiente sección.

8.3 Variaciones en formatos de instrucción

Las instrucciones se pueden clasificar de diversa manera en relación con su formato. Una clasificación común se basa en el número de direcciones en la instrucción (usualmente 0-39, aunque es posible tener más direcciones). Aquí, “dirección” se refiere a la especificación de un operando u otro valor y no necesita corresponder a una localidad en memoria. Por ejemplo, una especificación de registro, un operando inmediato o un *offset* relativo se califica como “dirección” en la siguiente discusión. Una base y un *offset*, que en conjunto especifican un solo operando de memoria, se cuentan como dos direcciones. De igual modo, un registro índice, cuando se especifica en una instrucción, se visualiza como una dirección separada aun cuando su contenido se pueda combinar con un valor base, e incluso un *offset*, para definir un solo operando de memoria.

Las **instrucciones de dirección cero** se deben apoyar en el direccionamiento implícito para sus operandos. Los ejemplos incluyen incrementar un registro implícito (acumulador), sacar el elemento alto de la *pila* en un registro implícito y saltar la siguiente instrucción en secuencia (con base en una condición implícita). MiniMIPS sólo tiene una de esas instrucciones: `syscall`.

Las **instrucciones de dirección uno** llevan una especificación de operando y usan direccionamiento implícito para cualquier otro operando necesario o destino de resultado. Los ejemplos incluyen sumar un valor al acumulador, empujar un valor especificado en la *pila* o saltar a una localidad específica. Tales instrucciones fueron predominantes en las primeras computadoras digitales. La instrucción *jump* *j* es un ejemplo de instrucciones de dirección uno en MiniMIPS.

Las **instrucciones de dirección dos** permiten operaciones aritméticas/lógicas de la forma $a = g(b)$, que involucren un operador unario, o $a = f(a, b)$, donde el destino de resultado es el mismo que uno de los operandos de un operador binario. De manera alterna, pueden haber tres operandos distintos como en las instrucciones de dirección tres, pero con una de ellas implícita. La instrucción `mult` de MiniMIPS cae dentro de esta última categoría. También es posible que una dirección especifique un solo operando, como en las instrucciones de dirección uno, y la otra dirección se usa para complementar la primera como un índice u *offset*. En el último caso, las dos “direcciones” en la instrucción son componentes de una sola especificación de operando. Se dice que tales instrucciones a veces pertenecen al formato dirección uno más uno.

Las **instrucciones de dirección tres** son comunes en los procesadores modernos cuyas instrucciones aritméticas/lógicas toman la forma $a = f(b, c)$. De nuevo es posible tener menos operandos distintos, y los campos *address* adicionales para direccionamiento más complejo. Por ejemplo, puede haber dos operandos correspondientes al cálculo $a = f(a, b)$, donde el operando *b* se especifica con el direccionamiento base o indexado. En MiniMIPS existen muchos ejemplos de instrucciones de dirección tres.

Aun cuando las instrucciones con cuatro o más direcciones son factibles y ocasionalmente se usan (vea la instrucción `ADDP6` de VAX en la tabla 8.1), son demasiado raras en las arquitecturas modernas como para garantizar una discusión adicional en este libro. La figura 8.2 muestra un ejemplo de instrucción MiniMIPS para cada una de las cuatro categorías descritas al principio de esta sección.

Categoría	Formato	Opcode	Descripción de operando(s)
Dirección 0	<div><div>0</div><div></div><div>12</div></div>	<code>syscall</code>	Un operando implícito en el registro <code>\$v0</code>
Dirección 1	<div><div>2</div><div>Dirección</div></div>	<code>j</code>	Salta a blanco direccionado (en forma pseudodirecta)
Dirección 2	<div><div>0</div><div>rs</div><div>rt</div><div></div><div>24</div></div>	<code>mult</code>	Dos registros fuente direccionadas, destino implícito
Dirección 3	<div><div>0</div><div>rs</div><div>rt</div><div>rd</div><div></div><div>32</div></div>	<code>add</code>	Destino y dos registros fuentes direccionados

Figura 8.2 Ejemplos de instrucciones MiniMIPS con 0 a 3 direcciones; las áreas sombreadas no se usan.

Tipo	Formato (anchos de campo mostradas)	Opcode	Descripción de operando(s)
1-byte	<div><div>5</div><div>3</div></div>	<code>PUSH</code>	Especificación de registro 3-bit
2-byte	<div><div>4</div><div>4</div><div>8</div></div>	<code>JE</code>	Condición 4-bit, salto de <i>offset</i> de 8-bit
3-byte	<div><div>6</div><div></div><div>8</div><div>8</div></div>	<code>MOV</code>	registro/modo 8-bit, <i>offset</i> 8-bit
4-byte	<div><div>8</div><div>8</div><div>8</div><div>8</div></div>	<code>XOR</code>	registro/modo 8-bit, base 8-bit/index, <i>offset</i> 8-bit
5-byte	<div><div>4</div><div>3</div><div>32</div></div>	<code>ADD</code>	registro 32 bi, intermedio 32-bit
6-byte	<div><div>7</div><div></div><div>8</div><div>32</div></div>	<code>TEST</code>	registro 8-bit/modo, intermedio 32-bit

Figura 8.3 Ejemplo de instrucciones 80×86 que varían en ancho desde 1 hasta 6 bytes; también existen instrucciones mucho más anchas (de hasta 15 bytes).

Todas las instrucciones en minimipis tienen 32 bits de ancho. Esta uniformidad, que simplifica muchos aspectos de implementación de hardware, es consecuencia de limitar los modos de direccionamiento. Por ejemplo, las instrucciones de dirección tres pueden especificar sólo tres registros, o dos registros más un operando inmediato. El uso de modos de direccionamiento complejos y flexibles con frecuencia necesitan formatos de instrucción variables para evitar ineficiencia extrema que resulte de los campos no usados en muchas instrucciones que no necesitan direcciones de memoria múltiples.

La arquitectura de conjunto de instrucciones Intel 80x86, usada por los procesadores Pentium y sus predecesores, es ejemplo de las instrucciones de ancho variable, que varían en ancho de 1 a 15 bytes. Se proporcionó todavía mayor variabilidad en el conjunto de instrucciones de la serie Digital VAX de las computadoras a finales de las décadas de 1970 y 1980 (para ejemplo de instrucciones, vea la tabla 8.1). Las ventajas e inconvenientes de los modos de direccionamiento complejos y los formatos de instrucción variable asociados, son similares a las que se citan para las instrucciones complejas. En las secciones 8.4 y 8.5 se elaborará en torno a las negociaciones en el diseño de conjuntos de instrucciones.

La figura 8.3 muestra algunos de los formatos más comunes en el ahora ampliamente usado conjunto de instrucciones Intel 80x86. Las instrucciones muestran rangos desde un solo byte (que es o sólo un *opcode* o un *opcode* más una especificación de registro de tres bits) hasta seis bytes. En todos los casos, el campo de la extrema izquierda de la instrucción, que varía en ancho desde cuatro hasta seis bits, especifica el *opcode*. Desde luego, los *opcode* se elegirán de modo que los más largos no

tengan alguno de los más cortos como prefijo. El hardware puede determinar, mediante inspección del primer byte de una instrucción 80x86, cuántos otros bytes se deben leer y procesar para completar la instrucción. El campo de un bit que se muestra en algunas de las instrucciones sirven a funciones cuya descripción no es competencia de este libro.

■ 8.4 Diseño y evolución del conjunto de instrucciones

En los primeros días de la computación digital se construyeron desde cero nuevas máquinas, comenzando con la elección de un conjunto de instrucciones. A través de los años, ciertas instrucciones probaron su utilidad y se incluyeron en muchos conjuntos de instrucciones. Sin embargo, éstos permanecieron incompatibles. Los programas en lenguaje de máquina y la propia máquina no eran portátiles, incluso entre máquinas construidas por la misma compañía, asimismo, los programas de alto nivel tuvieron que recompilarse para su ejecución en cada nueva máquina. La introducción, a mediados de la década de 1960, de la familia de computadoras IBM System/360, y su subsecuente éxito comercial, cambiaron el escenario. Con esta familia, que compartió la misma arquitectura de conjunto de instrucciones pero incluían máquinas con capacidades y rendimientos muy diferentes, se hizo posible correr los mismos programas en muchas computadoras. El amplio éxito comercial de la computadora personal de IBM, introducida a principios de la década de 1980, y la acompañante inversión en software de sistema y aplicaciones para ella, cimentó el dominio del conjunto de instrucciones x86 de Intel, así como el idioma inglés se convirtió en un estándar internacional *de facto* para los negocios y la comunicación técnica.

Si se le proporciona una pizarra en blanco, un arquitecto de computadoras puede diseñar un conjunto de instrucciones con las siguientes características deseables:

Consistencia, con reglas uniformes y generalmente aplicables.

Ortogonalidad, con características independientes que no interfieren.

Transparencia, sin efectos colaterales visibles debido a detalles de implementación.

Facilidad de aprender/uso (con frecuencia un subproducto de los tres atributos precedentes).

Extensibilidad, para permitir la adición de capacidades futuras.

Eficiencia, en términos tanto de necesidades de memoria como de realización en hardware.

Éstas son características generalmente reconocidas de un buen conjunto de instrucciones, pero, al igual que en muchos otros contextos técnicos, la bondad no se puede definir o evaluar sin referencia a las metas de diseño y aplicaciones pretendidas. Con los años, con el crecimiento del software en importancia y costo en relación con el hardware, la estética del diseño del conjunto de instrucciones propició el requisito de compactibilidad como el foco principal en el desarrollo de hardware. En la actualidad, los conjuntos de instrucciones rara vez se diseñan desde cero. Más bien, los conjuntos de instrucciones nuevos se forman mediante la modificación o extensión de los ya existentes. Este enfoque incremental, esencial para mantener la compatibilidad retrospectiva, protege las grandes inversiones en software y personal al asegurar que el software previamente desarrollado pueda correr en hardware nuevo.

La figura 8.4 muestra el proceso de desarrollo de un nuevo procesador. La tarea se asigna a un equipo de diseño del procesador formado por arquitecto jefe y decenas, o incluso cientos, de miembros con varias subespecialidades, desde diseño VLSI o tecnologías de implementación hasta evaluación de rendimiento y diseño de compilador. Si la meta del proyecto es introducir una versión un poco más rápida de un procesador existente, acaso con algunos *bugs* (problemas) descubiertos fijos, entonces se toma la trayectoria más baja, que involucra rediseño mínimo seguido de implementación o fabricación. En este contexto, la introducción de un nuevo procesador mayor, o mejora significativa de rendimiento para un producto existente, requiere un proceso de diseño más extenso y largo, que con frecuencia abarca muchos años.

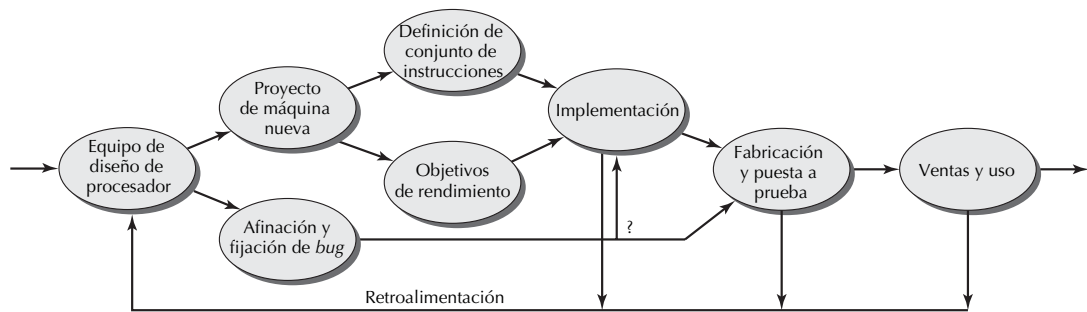


Figura 8.4 Diseño de un procesador y su proceso de implementación.

■ **TABLA 8.2** Evolución del conjunto de instrucciones y arquitectura x86 en algunos de los microprocesadores Intel, y productos equivalentes de AMD.

Intel	Año	Conjunto de instrucciones y otras modificaciones clave	AMD	Año
8086-8088	1978	Divergía del 8085 y sus predecesores y comenzó una nueva arquitectura de conjunto de instrucciones conocidos como x86		
80286	1982	Continuó con una arquitectura de 16 bits y permaneció completamente compatible hacia atrás con 8086-8088		
80386	1985	Introdujo la arquitectura extendida de 32 bits <i>dubbed</i> (aislada) IA-32, más tarde se aumentó con el coprocesador numérico 80387	Am386	1990
80486	1989	Integró el 80386 con su coprocesador numérico 80387 y una memoria caché en chip	Am486	1994
Pentium	1993	Introdujo un bus de datos de 64 bits junto con otros impulsores de rendimiento tal como conflicto de instrucción múltiple	K5	1996
Pentium II	1997	Incorporó 57 nuevas instrucciones de máquina que constituían las características de extensión multimedia (MMX)	K6	1997
Pentium III	1999	Incorporó 70 nuevas instrucciones de máquina que constituían la característica de <i>streaming</i> (transferencia de datos) de extensiones SIMD (SSE)	Athlon	1999
Pentium 4	2000	Incorporó una versión más avanzada de característica de <i>streaming</i> de extensiones SIMD (SSE2) y buses internos más amplios	Athlon XP	2001

La tabla 8.2 ejemplifica el enfoque evolutivo al diseño de conjunto de instrucciones. La arquitectura de conjunto de instrucciones de los procesadores más populares actualmente ofrecidos por Intel y AMD para aplicaciones de escritorio tiene sus orígenes en los microprocesadores Intel 8086 y 8088 de hace un cuarto de siglo. Este conjunto de instrucciones se extendió gradualmente para ajustar el movimiento de arquitecturas de 16 a 32 bits y para permitir el procesamiento eficiente de aplicaciones multimedia basadas en Web. Está en progreso una migración de arquitecturas a 64 bits. Intel introdujo una arquitectura completamente nueva IA-64, con palabras de instrucción largas y paralelismo explícito (operaciones múltiples empaçadas en una palabra de instrucción), mientras que AMD sigue x86-64, una extensión de 64 bits compatible hacia atrás de la arquitectura de conjunto de instrucciones x86/IA-32

■ **8.5 La dicotomía RISC/CISC**

Conforme las computadoras crecieron en complejidad, los conjuntos de instrucciones limitados de las primeras computadoras digitales dieron paso a conjuntos de operaciones enormemente complejos con una rica variedad de modos de direccionamiento. Entre las décadas de 1940 y 1970, se pasó de un mínimo de instrucciones a cientos de ellas diferentes, en una diversidad de formatos que condujo a miles de distintas combinaciones cuando se tomaron en cuenta las variaciones en modo de direccionamiento.

Sin embargo, ya en 1971 se reconoció que sólo un pequeño subconjunto de combinaciones disponibles se usaban en los programas en lenguaje de máquina generados manual o automáticamente, ello condujo a la conclusión de que quizá serían suficientes 32-64 operaciones [Fost71]. Con la llegada de VLSI, la simplificación de conjuntos de instrucciones y modos de direccionamiento, que ya habían probado bastante éxito en el mejoramiento del rendimiento de las supercomputadoras CDC y Cray, ganó impulso a principios de la década de 1980. El acrónimo RISC, para *Reduced Instruction-Set Computer* (computadora con conjunto de instrucciones reducido) [Patt82], ayudó a popularizar esta tendencia.

Parecía que los conjuntos de instrucciones se habían vuelto tan complicados que la inclusión de mecanismos para interpretar todas las combinaciones posibles de *opcodes* y operandos estaba frenando incluso las operaciones muy simples. Los primeros chips VLSI no tuvieron suficiente espacio para acomodar todos los circuitos de control, que tomaron casi la mitad del área de chip, y también retuvo el gran número de registros requeridos para un buen rendimiento. Para superar este problema de las computadoras con conjunto de instrucciones complejas, o CISC, se diseñaron y construyeron algunas máquinas de tipo RISC. La serie MIPS de procesadores (en la que se basa la máquina MIPS de los capítulos 5 a 7) y el procesador SPARC de Sun Microsystem fueron dos de los primeros productos en esta categoría. Las máquinas RISC bajaron el uso de control de 50% del área de chip a menos de 10%, y abrió espacio para más registros, así como mayores cachés en chip. Más aún, la simplicidad de control permitió una realización completamente alambrada para sustituir los entonces comunes controles microprogramados, ello condujo a una rapidez de ejecución que desplazó los efectos de usar muchas instrucciones para lograr los efectos de una instrucción compleja. Este último enunciado tendrá más sentido después de leer la parte cuatro del libro.

Desde luego, “reducido” y “complejo” son términos relativos, y algunos procesadores son difíciles de categorizar como RISC o CISC. Por tanto, RISC se debe ver como una forma de pensar correspondiente a la filosofía “pequeño es hermoso”, y un enfoque al diseño de conjunto de instrucciones, más que una clase de máquinas con fronteras concretas. Los diseños de procesador se adhieren a la filosofía RISC en diversos grados, lo cual forma casi un continuo desde las RISC puras en un extremo hasta las CISC absolutas en el otro. Las características que definen las RISC incluyen las siguientes:

1. Pequeño conjunto de instrucciones, cada una de las cuales se puede ejecutar en casi la misma cantidad de tiempo usando el control alambrado; una instrucción RISC corresponde a una *microinstrucción*, que se implementa de una secuencia de pasos en la ejecución de instrucciones más complejas (sección 14.5).
2. La arquitectura *load/store* que confina los cálculos de dirección de memoria y demoras de acceso a un pequeño conjunto de instrucciones *load* y *store*, y todas las otras obtienen sus operandos de registros más rápidos, y más compactamente direccionables (un número más grande de ellas se pueden acomodar debido a simplicidad de control).
3. Modos de direccionamiento limitado que eliminan o aceleran los cálculos de dirección para la gran mayoría de casos que se encuentran en la práctica y que incluso ofrecen suficiente versatilidad para permitir la síntesis de otros modos de direccionamiento en muchos pasos. Observe que, debido a la arquitectura *load/store*, una instrucción necesita cálculo de dirección o manipulación de operandos de registro, pero nunca ambos.
4. Formatos de instrucción simples y uniformes que facilitan la extracción/decodificación de los diversos campos y permiten traslape entre interpretación *opcode* y lectura de registro. Las instrucciones uniformes de ancho de palabra simplifican el prefetching de instrucción y eliminan complicaciones que surgen de instrucciones que cruzan fronteras de palabra, bloque o página.

Estas cuatro características conducen a una ejecución más rápida de las operaciones más usadas en el software. Desde luego, todos concuerdan en que más rápido es mejor; de este modo, ¿por qué se tienen

CISC? Existen dos ventajas potenciales de las CISC. Primero, economía en el número de instrucciones utilizadas (programas más compactos) traducidos a requerimiento de espacio de almacenamiento más pequeño, menores accesos de memoria y uso más efectivo del espacio caché; todo esto puede ser benéfico, pues la memoria se ha convertido en un grave problema en las computadoras modernas. Segundo, correspondencia más cercana entre instrucciones de máquina y operaciones o constructos que se encuentran en los lenguajes de alto nivel puede facilitar la compilación del programa, así como comprensión, corrección de problemas y modificación del código resultante.

En cierta forma, el estilo CISC constituye un resultado natural de la forma en que los conjuntos de instrucciones se expanden y del deseo por mantener compatibilidad hacia atrás (habilidad para correr programas existentes). Algunos diseños que comienzan como RISC puro crecen más complejos a lo largo del tiempo con la adición de nuevas “características” para ajustar las necesidades de aplicaciones emergentes. Si el conjunto de instrucciones de una máquina se vincula a su vocabulario, el problema se vuelve aparente. Suponga que quiere expandir un lenguaje natural para acomodar palabras propuestas para nuevas nociones y artilugios. Si la gente que habla el lenguaje fuera capaz de leer libros ya publicados, que a su vez se refieren como libros más viejos, etc., usted no sería capaz de eliminar alguna palabra existente que sea obsoleta o incompatible con nuevos descubrimientos. Imagine cuán gracioso sonaría el español moderno, y cuánto más grueso sería el diccionario escolar típico, si fuese requisito comprender toda palabra de español que ya se haya usado.

Así que, al final, ¿qué es mejor: RISC o CISC? Usted obtendrá diferentes respuestas a esta pregunta dependiendo de a quién le pregunte o cuál libro lea. Ciertamente, RISC es más fácil de describir en un libro de texto o en clase. En este sentido, la exposición de este libro se basa en MiniMIPS, una simple máquina RISC. Más allá de lo anterior no se puede formular un enunciado general. A partir de la discusión de rendimiento en el capítulo 4, se sabe que, lo que a final de cuentas importa, es el tiempo de corrida de los programas de aplicación; no importa si una computadora ejecuta diez mil millones de instrucciones simples o cinco mil millones de instrucciones complejas, sino cuánto tiempo tarda en realizar las tareas requeridas.

Sin embargo, si todo esto es igual, RISC tiene una ventaja innegable. Diseñar y construir un procesador consumen tiempo y conlleva gastos. Literalmente tarda años y la participación de cientos de diseñadores de hardware para desarrollar el siguiente modelo de un procesador CISC. Mucho de este esfuerzo se dedica a la verificación y funcionamiento de la prueba del diseño. El enfoque RISC simplifica el diseño y desarrollo de un nuevo procesador, por ello reduce los costos de desarrollo y acorta el tiempo de aparición en el mercado de los nuevos productos. En el clima empresarial actual, éstos son beneficios importantes. Además, si se construye un sistema en un chip será más probable que un procesador RISC embone en el chip con todos los otros subsistemas requeridos y sea más fácil de integrar y poner a prueba.

En conclusión, se puede demostrar por qué el enfoque RISC puede ser benéfico en términos de la ley de Amdahl.

Ejemplo 8.1: Comparación RISC/CISC mediante la fórmula de Amdahl generalizada Una arquitectura de conjunto de instrucciones tiene dos clases de instrucciones simples (S) y complejas (C). En una implementación de referencia existente de la ISA, las instrucciones de clase S representan 95% del tiempo de corrida para los programas de interés. Se considera que una versión RISC de la máquina sólo ejecuta instrucciones de clase S directamente en hardware, y las instrucciones de clase C se tratan como seudoinstrucciones (convertidas a secuencias de instrucciones de clase S por el ensamblador). Se estima que, en la versión RISC, las instrucciones de clase S correrán 20% más rápido y las instrucciones de clase C se frenarán por un factor de 3. ¿El enfoque RISC ofrece mejor o peor rendimiento que la implementación de referencia?

Solución: Se usa una forma generalizada de la fórmula de aceleración de Amdahl que produzca la aceleración global cuando 0.95 del trabajo se acelere por un factor de $1.0/0.8 = 1.25$, mientras que el restante 5% se frena por un factor de 3. La aceleración RISC global es $1/[0.95/1.25 + 0.05 \times 3] = 1.1$. En consecuencia, se puede esperar 10% de mejoría en rendimiento en la versión RISC propuesta de la máquina. Esto es muy significativo, pues se ahorra en costo y tiempo en diseño, funcionamiento y fabricación.

Es posible combinar las ventajas de RISC y CISC en una sola máquina. De hecho, los actuales procesadores CISC de alto rendimiento, de los que los dispositivos en la serie Intel Pentium son ejemplo de primera clase, se diseñan con el uso de un traductor de hardware frontal que sustituye cada instrucción CISC con una secuencia de una o más operaciones análogas a CISC. Entonces estas operaciones más simples se ejecutan con gran rapidez, usando todos los beneficios y métodos de aceleración aplicables a los conjuntos de instrucciones del tipo RISC.

8.6 Dónde dibujar la línea

Al haber aprendido acerca de las máquinas RISC con unas cuantas docenas de instrucciones, usted puede preguntar si existe un límite a la simplicidad de un conjunto de instrucciones y si ir por abajo de tal límite comprometería las capacidades computacionales de la máquina. En otras palabras, ¿cuántas instrucciones tiene lo último en RISC (URISC)? Se evidencia que, si se ignoran entrada/salida, interrupciones y otras instrucciones de calendarización y contabilidad que se necesitan en cualquier computadora, una sola instrucción sería adecuada; esto es, desde el aspecto informático se puede obtener una computadora con sólo una instrucción. Esto es bastante sorprendente. Una consecuencia de aprender acerca de esta única instrucción es que se deben incluir instrucciones adicionales en un conjunto de instrucciones por una buena causa; esto es, sólo si se puede verificar que ayudan, más que impiden, el rendimiento de la máquina.

Vea si se puede argumentar cuál puede ser la instrucción única. Primero, note que la instrucción no necesita un *opcode*. Para ser capaz de realizar aritmética, necesita dos operandos y un resultado; haga el destino de resultado igual al segundo operando. Los operandos tienen que provenir de memoria y el resultado se debe almacenar de vuelta en memoria, dado que no se cuenta con instrucciones separadas para cargar en, o almacenar de, registros. De este modo, hasta el momento se necesitan dos direcciones de memoria en la instrucción: fuente 1 y fuente 2/destino. Desde luego, no se puede lograr nada muy útil sin una capacidad de bifurcación para permitir cálculos y ciclos condicionales dependientes de datos. Así que se postula que la sola instrucción incorporará una bifurcación condicional que requiere una tercera dirección para especificar el destino de la bifurcación. ¿Y qué hay acerca de la condición *branch*? Si se usa la sustracción como operación aritmética, se puede tomar el signo del resultado como condición de bifurcación, que no requiere especificación adicional para la condición.

Lo anterior lleva a la siguiente instrucción para URISC:

```
restar operand1 de operand2, sustituir operand2 con el
resultado, y saltar a la dirección blanco en caso
de resultado negativo
```

Aunque no se necesita un *opcode*, esta instrucción única se puede escribir en un formato similar al del lenguaje ensamblador MiniMIPS con el uso de `urisc` como el único *opcode*:

```
label: urisc dest,src1,target
```

Se usarán las convenciones de que la ejecución del programa siempre comienza en la ubicación de memoria 1 y que la bifurcación a la ubicación 0 detiene la ejecución del programa. Finalmente, se usa la directiva de ensamblador `.word` para nombrar e inicializar una palabra de memoria a usar como operando. El siguiente fragmento de programa copia los contenidos de la localidad de memoria `src` a la localidad de memoria `dest`; por tanto, es equivalente a la seudoinstrucción MiniMIPS `move`; usa otra localidad `temp` para almacenamiento temporal.

```
stop: .word 0
start: urisc dest,dest,+1    # dest = 0
      urisc src,dest,+1     # temp = -(src)
      urisc temp,dest,+1    # dest = -(temp)
      ...                  # resto del programa
```

Observe que, en lenguaje ensamblador, todo programa comienza con la directiva `.word 0` para permitir la terminación adecuada. Además, para evitar tener que etiquetar cada instrucción en el programa, se usa la notación “ $\pm i$ ” en el campo *branch target address* (dirección destino de bifurcación) para dar a entender que el destino de bifurcación está a i instrucciones adelante o atrás.

Ejemplo 8.2: Seudoinstrucciones URISC Para cada una de las siguientes seudoinstrucciones, escriba las instrucciones correspondientes producidas por ensamblador URISC. El prefijo “u” se usa para distinguir este tipo de instrucciones del que se refiere a las instrucciones MiniMIPS.

```
parta: uadd dest,src1,src2    # dest = (src1)+(src2)
partb: uswap src1,src2        # intercambia (src1) y (src2)
partc: uj label               # ir a label
partd: ubge src1,src2,label   # si (src1) ≥ (src2), ir a label
parte: ubeq src1,src2,label   # si (src1) = (src2), ir a label
```

Solución: En lo que sigue, `at1` y `at2` son localidades de memoria temporales para uso exclusivo del ensamblador (similar al papel del registro `$at` en MiniMIPS) y `one` es una localidad de memoria que retiene la constante 1.

```
parta: urisc at1,at1,+1      # at1 = 0
      urisc src1,at1,+1     # at1 = -(src1)
      urisc src2,at1,+1     # at1 = -(src1) - (src2)
      urisc dest,dest,+1    # dest = 0
      urisc at1,dest,+1     # dest = -(at1); esto es, (src1)+(src2)
partb: urisc at1,at1,+1      # at1 = 0
      urisc at2,at2,+1      # at2 = 0
      urisc src1,at1,+1     # at1 = -(src1)
```

```

urisc src2,at2,+1 # at2 = -(src2)
urisc src1,src1,+1 # src1 = 0
urisc src2,src2,+1 # src2 = 0
urisc at2,src1,+1 # src1 = -(at2); esto es, (src2)
urisc at1,src2,+1 # src2 = -(at1); esto es, (src1)

partc: urisc at1,at1,+1 # at1 = 0
       urisc one,at1,label # at1 = -1, para forzar jump

partd: urisc src2,src1,+3 # si (src1)-(src2) < 0, skip 2 instr's
       urisc at1,at1,+1 # at1 = 0
       urisc one,at1,label # at1 = -1, para forzar jump

parte: urisc src2,src1,+4 # si (src1)-(src2) < 0, skip 3 instr's
       urisc src1,src2,+3 # si (src2)-(src1) < 0, skip 2 instr's
       urisc at1,at1,+1 # at1 = 0
       urisc one,at1,label # at1 = -1, para forzar jump

```

En la solución para parte, se usó el hecho de que $x = y$ si $x \geq y$ y $y \geq x$.

La figura 8.5 muestra los elementos necesarios para implementar URISC en hardware. Existen cuatro registros que incluyen las direcciones de memoria y los registros de datos (MAR, MDR), un sumador, dos banderas de condición N y Z (negativo, cero) que caracterizan la salida del sumador, y una unidad de memoria. Los pequeños círculos sombreados representan puntos de control, y la señal de control que permite que suceda la transferencia de datos se escribe al lado del círculo. Por ejemplo, la señal de control PC_{in} , cuando se postula, causa que la salida del sumador se almacene en el contador del programa.

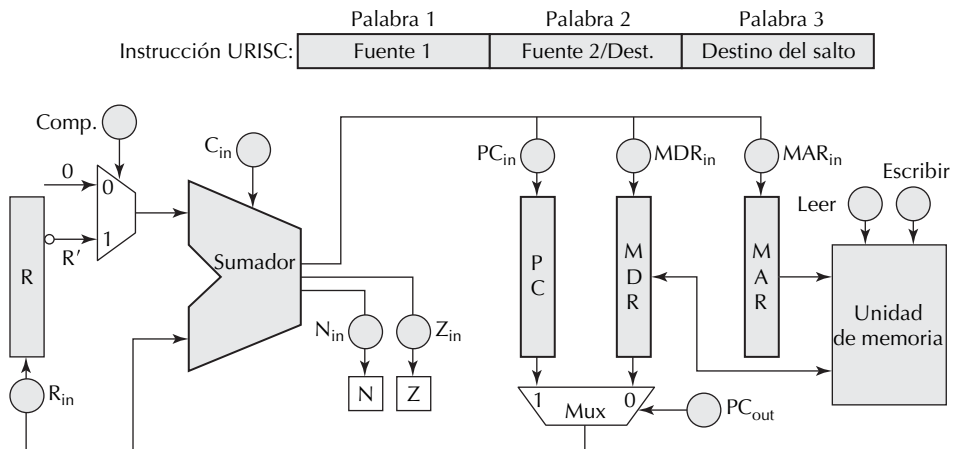


Figura 8.5 Formato de instrucción y estructura de hardware para URISC.

PROBLEMAS

8.1 Instrucciones complejas

Para cada instrucción compleja que se menciona en la tabla 8.1, estime el número de instrucciones MiniMIPS que se necesitarían para lograr el mismo efecto. Se proporcionan la *cuenta estática* (número de instrucciones de máquina generadas) y la *cuenta dinámica* (número de instrucciones de máquina ejecutadas). Las 13 líneas de la tabla 8.1 constituyen las partes a–m de este problema.

8.2 Formatos de instrucción

Categorice cada una de las instrucciones MiniMIPS de la tabla 6.2, según el número de direcciones en su formato (instrucción de dirección 0, 1, 2 o 3, como en la figura 8.2). Para efectos de este problema, una dirección representa cualquier conjunto de uno o más campos que especifican explícitamente el paradero de un operando distinto, esté en registro o en memoria.

8.3 Formatos de instrucción

¿Tiene sentido hablar acerca de seudoinstrucciones de dirección 0, 1, 2 y 3, como se hace acerca de las instrucciones? Límite su discusión en términos de las seudoinstrucciones MiniMIPS que se muestran en la tabla 7.1.

8.4 Arquitectura frente a implementación

Clasifique cada uno de los siguientes objetos como pertenecientes a la arquitectura de conjunto de instrucciones, el nivel usuario/aplicación arriba de ella, al nivel de implementación debajo de ella. Justifique sus respuestas.

- Formato de representación de número.
- Mapa *opcode* (tabla que menciona la codificación binaria de cada *opcode*).
- Convenciones de uso de registro, como en la figura 5.2.
- Modos de direccionamiento.
- Sumador con anticipación de acarreo.
- Mapa de memoria, como en la figura 6.3.
- Codificación de caracteres para cadenas, como en la tabla 6.1 (ASCII).

8.5 Atributos del conjunto de instrucciones

Evalúe el conjunto de instrucciones MiniMIPS que se muestra en la tabla 6.2 en relación con los atributos de

un buen conjunto de instrucciones enumerados en la sección 8.4.

8.6 Atributos de buenas interfases de usuario

Los atributos de un buen conjunto de instrucciones, enumerados en la sección 8.4, también se aplican a otras interfases de usuario. Abra cada uno de los siguientes tipos de sistema, considere un par de diferentes instancias con las que usted esté familiarizado y discuta cómo se comportan sus interfases de usuario en relación con dichos atributos respecto de los otros.

- Elevadores
- Cajeros automáticos
- Controles remoto de televisión
- Relojes de alarma
- Teléfonos (fijos o móviles)
- Calculadoras
- Tiendas en internet

8.7 Codificaciones del conjunto de instrucciones

Se va a diseñar una máquina cuyas instrucciones varían desde uno hasta seis bytes de ancho, con instrucciones de dos bytes como las más comunes e instrucciones de cinco y seis bytes como las menos usadas. Compare los siguientes esquemas de codificación para el campo *opcode* de las instrucciones.

- Los tres bits de la extrema izquierda en el primer byte de cada instrucción contiene un número binario en [1, 6] que indica el ancho de la instrucción en bytes.
- Instrucciones de un byte que comienzan con 0 en la posición de bit de la extrema izquierda del primer byte, instrucciones de dos bytes que comienzan con diez, y todas las otras instrucciones que comienzan con 11 seguidas por un campo de dos bits que indican el número de bytes adicionales (más allá del tercero) en la instrucción.

8.8 El contador de programa

En MiniMIPS, el contador de programa constituye un registro dedicado que siempre retiene la dirección de la siguiente instrucción que se ejecutará. Algunos procesadores insertan el contador del programa en el archivo de registro general de modo que se puede tratar como

cualquier otro registro. Por ejemplo, en MiniMIPS se puede designar el registro \$25 como \$pc en lugar de \$t9 (figura 5.2).

- Mencione dos efectos positivos de este cambio. Justifique sus respuestas.
- Mencione dos efectos negativos de este cambio. Justifique sus respuestas.

8.9 Pila de hardware

En la sección 8.2 se menciona que se han construido ciertas máquinas que predominantemente usan instrucciones de dirección 0. Esas máquinas sacan sus operandos requeridos de lo alto de una pila y empujan sus resultados a lo alto de la misma *pila*. Si ésta es una región en memoria, tales instrucciones serían bastante lentas, porque cada una requeriría entre dos y tres accesos a memoria de datos. Al proporcionar un diseño lógico completo, muestre cómo se puede diseñar una *pila* de hardware de 33 entradas para acelerar los accesos de datos. Use un registro para retener el elemento superior de la *pila* y 32 registros de corrimiento bidireccionales, cada uno con 32 bits de ancho, para retener los restantes 32 elementos. De esta forma, los dos elementos superiores de la *pila* son fácilmente accesibles, y las operaciones de empujar y sacar se pueden realizar a través del corrimiento de los registros. Ignore la posibilidad de desbordamiento de pila (empujar un nuevo elemento cuando la *pila* está llena) o subdesbordamiento (intentar sacar un elemento cuando la *pila* está vacía).

8.10 Operandos basados en memoria

Se contempla la adición de un nuevo conjunto de instrucciones a MiniMIPS que realizan operaciones aritméticas y corrimiento/lógicas sobre el contenido de un registro y una localidad de memoria, y almacenan el resultado en el mismo registro. Por ejemplo, `addm $t0, 1000($s2)` lee el contenido de la localidad de memoria `1000+($s2)` y lo suma al contenido del registro `$t0`. Cada una de dichas instrucciones puede eliminar la necesidad de una instrucción del tipo *load*. En el lado negativo esta instrucción más compleja alargaría el ciclo de reloj de la máquina a 1.1 veces su valor original (suponga que no se afectan los CPI para las diversas clases de instrucción). Al considerar las distribuciones de uso de instrucción de la tabla 4.3, ¿bajo qué condiciones la modificación sugerida conduciría a un

rendimiento mejorado? En otras palabras, ¿qué fracción de las instrucciones *load* se debe eliminar para que el nuevo modo de direccionamiento sea benéfico?

8.11 MiniMIPS-64

Se diseñará un nuevo modelo de MiniMIPS con registros de 64 bits y unidad aritmética/lógica. Para mantener la compatibilidad retrospectiva con la actual versión de 32 bits, se decidió conservar todas las instrucciones actuales y hacerlas operar en las mitades inferiores de los registros respectivos, mientras se agregan nuevas versiones de 64 bits para tratar con registros completos. Con referencia a la tabla 6.2, conteste las preguntas siguientes.

- ¿Cuáles instrucciones requieren versiones de 64 bits?
- Sugiera modificaciones adecuadas de la codificación de instrucciones y la notación de lenguaje ensamblador para dar cabida a las nuevas instrucciones.
- ¿Qué problemas provoca el campo *immediate* de 16 bits y cómo propone manipularlos?
- ¿Puede identificar algún otro problema que resulta de esta extensión arquitectónica?

8.12 Fórmula de Amdahl generalizada en diseño ISA

Divida las instrucciones de un procesador en clases C_1, C_2, \dots, C_q , y sea f_i la fracción de tiempo gastado en la ejecución de las instrucciones de la clase C_i . Claramente, $f_1 + f_2 + \dots + f_q = 1$. Suponga que, comenzando desde una implementación de referencia, el tiempo de ejecución de las instrucciones de la clase C_i se mejora por un factor p_i , $1 \leq i \leq q$, y que $f_1 > f_2 > \dots > f_q$. Aún más, suponga que el costo total de las mejoras es proporcional a $p_1 + p_2 + \dots + p_q$. La siguiente forma generalizada de la ley de Amdahl denota la aceleración global:

$$S = \frac{1}{f_1/p_1 + f_2/p_2 + \dots + f_q/p_q}$$

Demuestre que, bajo estas condiciones, la inversión en hardware para acelerar las instrucciones de la clase C_i debe ser proporcional a f_i^2 para máxima aceleración global.

8.13 URISC

- Caracterice la instrucción única URISC con respecto a su formato y modo(s) de direccionamiento.

- b) ¿La instrucción única sería adecuada para acomodar operaciones I/O si se usa I/O mapeado por memoria?
Sugerencia: Revise la sección 22.2 para tomar algunas ideas.
- c) ¿Cuántas instrucciones MiniMIPS se necesitan para emular el efecto de una instrucción URISC única?
- d) Si su respuesta a la parte c) es x , ¿esto significa que MiniMIPS es x veces más lenta que URISC? Discuta.

8.14 Seudoinstrucciones URISC

Para cada una de las pseudoinstrucciones URISC del ejemplo 8.2, escriba una instrucción o secuencia de instrucciones MiniMIPS equivalente para lograr el mismo efecto.

8.15 URISC modificada

El diseño de URISC que se presentó en la sección 8.6 supone que números y direcciones de cada una ocu-

pan una palabra de memoria (por decir, 32 bits). Discuta cómo cambiarían las instrucciones y el diseño de URISC si las direcciones tuviesen la mitad del ancho de los números (por decir, 32 frente a 64 bits). Establezca claramente todas sus suposiciones.

8.16 Modos de direccionamiento para procesamiento de señales

Algunos procesadores digitales de señales implementan modos de direccionamiento especiales que aceleran ciertos tipos de cálculos. Estudie los siguientes tipos especiales de modo de direccionamiento y escriba un informe de dos páginas acerca de los aspectos de utilidad e implementación de cada uno.

- Direccionamiento circular, o de módulo (dirigido para buffer circulares).
- Direccionamiento de inversión de bit (dirigido para transformada rápida de Fourier).

REFERENCIAS Y LECTURAS SUGERIDAS

- [Blaa97] Blaauw, G. A. y F. P. Brooks Jr, *Computer Architecture: Concepts and Evolution*, Addison-Wesley, 1997.
- [Fost71] Foster, C., R. Gonter y E. Riseman, "Measures of Op-Code Utilization", *IEEE Trans. Computers*, vol. 20, núm. 5, pp. 582-584, mayo de 1971.
- [Giff87] Gifford, D. y A. Spector, "Case Study: IBM's System/360-370 Architecture", *Communications of the ACM*, vol. 30, núm. 4, pp. 292-307, abril de 1987.
- [Henn03] Hennessy, J. L. y D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 3a. ed., 2003.
- [Kane92] Kane, G. y J. Heinrich, *MIPS RISC Architecture*, Prentice Hall, 1992.
- [Mava88] Mavaddat, F. y B. Parhami, "URISC: The Ultimate Reduced Instruction Set Computer", *Int. J. Electrical Engineering Education*, vol. 25, pp. 327-334, 1988.
- [Patt82] Patterson, D. A. y C. H. Sequin, "A VLSI RISC," *IEEE Computer*, vol. 15, núm. 9, pp. 8-21, septiembre de 1982.
- [Patt98] Patterson, D. A. y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2a. ed., 1998.
- [Siew82] Siewiorek, D. P., C. G. Bell y A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, 1982.
- [Stal03] Stallings, W., *Computer Organization and Architecture*, Prentice Hall, 6a. ed., 2003.

PARTE TRES

LA UNIDAD ARITMÉTICA/LÓGICA

"Sólo tomé el curso regular." "¿Cuál fue?", preguntó Alicia. "Tambalearse y contorsionarse, por supuesto, para empezar replicó la tortuga de mentira y luego las diferentes ramas de la aritmética: ambición, distracción, afeamiento y mordacidad."

Lewis Carroll, Las aventuras de Alicia en el país de las maravillas

"...incluso en tal apretada, ordenada lógica, y si fuese un sistema completamente aritmético, es posible formular teoremas que no se pueden mostrar o como verdaderos o falsos".

J. Bronowski, El sentido común de la ciencia

TEMAS DE ESTA PARTE

- 9. Representación de números
- 10. Sumadores y ALU simples
- 11. Multiplicadores y divisores
- 12. Aritmética con punto flotante

Las manipulaciones de datos en una computadora digital se encuentran bajo las categorías de operaciones aritméticas (suma, resta, multiplicación, división) y la manipulación de bits u operaciones lógicas (NOT, AND, OR, XOR, corrimiento, rotación, etc.). Las operaciones lógicas son más o menos directas, pero instrumentar operaciones aritméticas requiere conocimiento de aritmética computacional, o sea de la rama del diseño de computadoras que trata con algoritmos para manipular números mediante circuitos de hardware o rutinas de software. La discusión de la aritmética computacional comienza con la representación de números. Esto último se debe a que la representación estándar de números en las computadoras es diferente de la que se usa para cálculos en pluma y papel o con calculadoras electrónicas. Además, la representación de números afecta la facilidad y rapidez de la aritmética en hardware y software. Lo anterior conduce a una diversidad de formatos no convencionales de representación de números. Aquéllos son, con frecuencia, invisibles al programador y se usan internamente (junto con sus algoritmos de conversión asociados para/desde formatos estándar) para mejoramiento de la rapidez.

Después de revisar en el capítulo 9 los esquemas más importantes de representación de números para computadoras digitales, en el capítulo 10 se describe el diseño de contadores, sumadores, unida-

des de operación lógica y corredores (componentes que posee toda computadora digital), los esquemas de multiplicación y división en el capítulo 11 y la aritmética de punto flotante en el capítulo 12. Los lectores interesados sólo en el diseño básico de ALU entera pueden omitir el estudio de las secciones 9.5 y 9.6 y los capítulos 12 y 13. Las discusiones generales en esta parte se relacionan para especificar instrucciones MiniMIPS donde sea aplicable.

REPRESENTACIÓN DE NÚMEROS

“Leibniz creyó ver la imagen de la creación en su aritmética binaria, en la que empleó sólo los dos caracteres, cero y unidad. Imaginó que la unidad puede representar a Dios, y el cero a la nada...”

Pierre Laplace, Teoría analítica de las probabilidades, 1812

“Esto no puede estar bien... ¡se va al rojo!”

Niño pequeño, cuando se le pide restar 36 a 24 (leyenda en una caricatura de artista desconocido)

TEMAS DEL CAPÍTULO

- 9.1 Sistemas numéricos posicionales
- 9.2 Conjuntos de dígitos y codificaciones
- 9.3 Conversión de base numérica
- 9.4 Enteros con signo
- 9.5 Números de punto fijo
- 9.6 Números de punto flotante

La representación de números es quizá el tema más importante en la aritmética computacional. Cómo se representen los números afecta la compatibilidad entre las máquinas y sus resultados computacionales e influye en el costo de implementación y la latencia de los circuitos aritméticos. En este capítulo se revisarán los métodos más importantes para representar enteros (magnitud con signo y complemento a 2) y números reales (formato ANSI/IEEE estándar de punto flotante). También se aprende acerca de otros métodos de representación de números, conversión de base numérica y codificación binaria de conjuntos de dígitos arbitrarios.

■ 9.1 Sistemas numéricos posicionales

Cuando se piensa en números, usualmente los *números naturales* son los que llegan primero a la mente; los números que secuencian libros o páginas de calendario, marcan las carátulas de reloj, parpadean en los marcadores de los estadios y guían las entregas a las casas. El conjunto $\{0, 1, 2, 3, \dots\}$ de números naturales, denominado *números enteros* o *enteros sin signo*, forman la base de la aritmética. Hace cuatro mil años, los babilonios sabían acerca de los números naturales y eran fluidos en aritmética. Desde entonces, las representaciones de los números naturales ha avanzado en paralelo con la

evolución del lenguaje. Las civilizaciones antiguas usaban varas y cuentas para registrar inventarios o cuentas. Cuando surgió la necesidad de números más grandes, la idea de agrupar varas o cuentas simplificó el conteo y las comparaciones. Con el tiempo, se usaron objetos de diferentes formas o colores para denotar tales grupos, lo que propició representaciones más compactas.

Los números se deben diferenciar de sus representaciones, a veces llamadas *numerales*. Por ejemplo, el número “27” se puede representar de diferentes formas mediante el uso de varios numerales o *sistemas de numeración*; éstos incluyen:

	varas o código <i>unario</i>
27	base 10 o código <i>decimal</i>
11011	base 2 o código <i>binario</i>
XXVII	numerales romanos

Sin embargo, no siempre se hace la diferencia entre números y numerales y con frecuencia se usan “números decimales” en lugar de “numerales decimales” para referirse a una representación en base 10.

Las bases distintas de 10 también aparecieron a través del tiempo. Los babilonios usaron números en base 60, ello hacía fácil tratar con el tiempo. Las bases 12 (*duodecimal*) y 5 (*quinaria*) también se han usado. El uso de números con base 2 (*binario*) se volvió popular con la aparición de las computadoras debido a su uso de dígitos binarios, o *bits*, que tienen sólo dos posibles valores, 0 y 1; además es compatible con las señales electrónicas. A los números de base 3 (*ternaria*) se les dio seria consideración en las primeras etapas del desarrollo de las computadoras digitales, pero los números binarios eventualmente ganaron. Los números con base 8 (*octal*) y base 16 (*hexadecimal*) se usan como notación abreviada para los números binarios. Por ejemplo, un número binario de 24 bits se puede representar como octal de ocho dígitos o uno hexadecimal de seis dígitos al tomar los bits en grupos de tres y cuatro, respectivamente.

En un *sistema numérico posicional* general de base r , con ancho de palabra fijo k , un número x se representa mediante una cadena de k dígitos x_i , con $0 \leq x_i \leq r - 1$.

$$x = \sum_{i=0}^{k-1} x_i r^i = (x_{k-1}x_{k-2} \cdots x_1x_0)_r$$

Por ejemplo, al usar base 2:

$$27 = (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = (11011)_{\text{dos}}$$

En un sistema numérico de base r y k dígitos, se pueden representar los números naturales de 0 a $r^k - 1$. Por el contrario, dado un rango de representación deseado $[0, P]$, el número requerido k de dígitos en base r se obtiene a partir de la ecuación:

$$k = \lceil \log_r(P + 1) \rceil = \lceil \log_r P \rceil + 1$$

Por ejemplo, la representación del número decimal 3125 requiere 12 bits en base 2, seis dígitos en base 5 y cuatro dígitos en base 8.

El caracter finito de las codificaciones numéricas en las computadoras tiene implicaciones importantes. La figura 9.1 muestra codificaciones binarias de cuatro bits para los números 0 al 15. Esta codificación no puede representar los números 16 y mayores (o -1 y menores). Es conveniente ver las operaciones aritméticas de suma y resta como el giro de la rueda dentada de la figura 9.1 en contrasentido a las manecillas del reloj y en sentido inverso, respectivamente. Por ejemplo, si la rueda dentada se fija de modo que la flecha vertical apunta al número 3 y luego se da vuelta en contrasentido a las manecillas del reloj mediante cuatro muescas para agregar el número 4, la flecha apuntará a la suma correcta 7. Si se intenta el mismo procedimiento con los números 9 y 12, la rueda se enredará pasando 0 y la flecha apuntará a 5, que es 16 unidades más pequeña que la suma correcta 21. Por ende, el re-

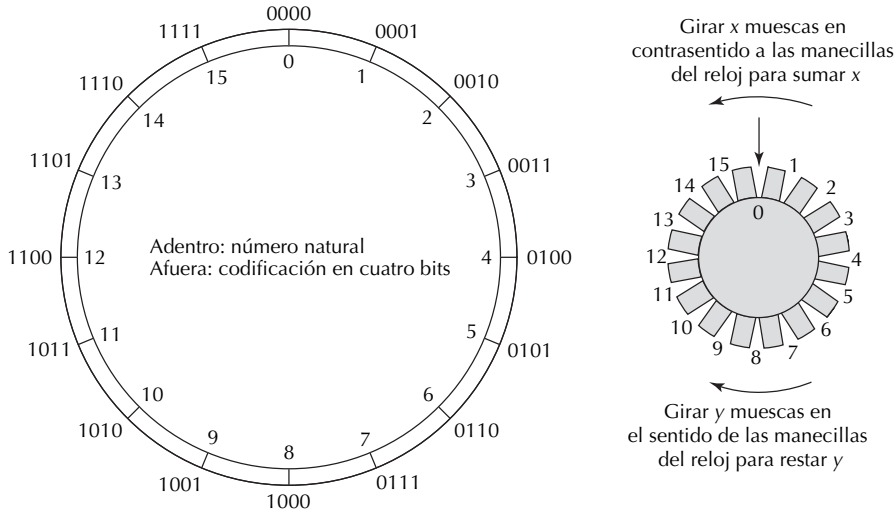


Figura 9.1 Representación esquemática de código de cuatro bits para enteros en $[0, 15]$.

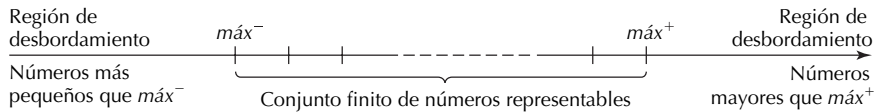


Figura 9.2 Regiones de desbordamiento en sistemas finitos de representación numérica. Para las representaciones sin signo cubiertas en esta sección, $máx^- = 0$.

sultado obtenido constituye la suma módulo 16 de 9 y 12 y ocurre un *desbordamiento* (*overflow*). Los desbordamientos ocasionales son inevitables en cualquier sistema finito de representación numérica.

Una situación similar surge en la resta $9 - 12$, porque el resultado correcto, -3 , no es representativo en la codificación de la figura 9.1. La interpretación de rueda dentada, en este caso, es establecer la rueda de modo que la flecha apunte a 9 y luego darle vuelta en el sentido de las manecillas del reloj por 12 muescas, ello pone la flecha en el número 13. Este resultado, que es 16 unidades más largo, representa de nuevo la diferencia en módulo 16 de 9 y 12. En este caso, se infiere que ha ocurrido un subdesbordamiento (*underflow*); sin embargo, este término en aritmética computacional se reserva para números muy pequeños en magnitud para ser distinguible de 0 (sección 9.6). Cuando el rango del sistema numérico se excede de su extremo inferior, todavía dice que ha ocurrido un *desbordamiento*. La figura 9.2 muestra las regiones de desbordamiento de un sistema finito de representación numérico.

Ejemplo 9.1: Sistemas numéricos posicionales de base fija Para cada uno de los siguientes sistemas de representación numérica convencional de base x , use los valores digitales del 0 al $r - 1$, determine el número más grande $máx$ que sea representativo con el número indicado k de dígitos y el número mínimo K de dígitos requeridos para representar todos los números naturales menores que un millón.

- $r = 2, k = 16$
- $r = 3, k = 10$
- $r = 8, k = 6$
- $r = 10, k = 8$

Solución: El número natural más grande a representar para la segunda parte de la pregunta es $P = 999\,999$.

- a) $\text{máx} = r^k - 1 = 2^{16} - 1 = 65\,535$; $K = \lceil \log_r(P + 1) \rceil = \lceil \log_2(10^6) \rceil = 20$ ($2^{19} < 10^6$)
- b) $\text{máx} = 3^{10} - 1 = 59\,048$; $K = \lceil \log_3(10^6) \rceil = 13$ ($3^{12} < 10^6$)
- c) $\text{máx} = 8^6 - 1 = 262\,143$; $K = \lceil \log_8(10^6) \rceil = 7$ ($8^6 < 10^6$)
- d) $\text{máx} = 10^8 - 1 = 99\,999\,999$; $K = \lceil \log_{10}(10^6) \rceil = 6$

Ejemplo 9.2: Desbordamiento en aritmética entera Para cada uno de los siguientes sistemas convencionales de representación numérica en base r , determine si evaluar la expresión aritmética mostrada conduce a desbordamiento. Todos los operandos se dan en base 10.

- a) $r = 2, k = 16; 10^2 \times 10^3$
- b) $r = 3, k = 10; 15\,000 + 20\,000 + 25\,000$
- c) $r = 8, k = 6; 555 \times 444$
- d) $r = 10, k = 8; 3^{17} - 3^{16}$

Solución: No es necesario representar los operandos involucrados en los sistemas numéricos especificados; más bien, se comparan los valores encontrados en el curso de la evaluación de la expresión con los máximos valores representativos derivados en el ejemplo 9.1.

- a) El resultado 10^5 es mayor que $\text{máx} = 65\,535$, de modo que ocurrirá desbordamiento.
- b) El resultado $60\,000$ es mayor que $\text{máx} = 59\,048$, de modo que ocurrirá desbordamiento.
- c) El resultado $246\,420$ no es mayor que $\text{máx} = 262\,143$, de modo que no se presentará desbordamiento.
- d) El resultado final $86093\,442$ no es mayor que $\text{máx} = 99\,999\,999$. Sin embargo, si se evalúa la expresión al calcular primero $3^{17} = 129\,140\,163$ y luego restar 3^{16} de ello, se encontrará desbordamiento y el cálculo no se puede completar correctamente. Al reescribir la expresión como $3^{16} \times (3 - 1)$ se elimina la posibilidad de un desbordamiento innecesario.

■ 9.2 Conjuntos de dígitos y codificaciones

Los dígitos de un número binario se pueden representar directamente como señales binarias, que luego se pueden manipular mediante circuitos lógicos. Las bases mayores que 2 requieren conjuntos de dígitos que tienen más de dos valores. Tales conjuntos de dígitos se deben codificar como cadenas binarias para permitir su almacenamiento y procesamiento dentro de circuitos lógicos convencionales de dos valores. Un conjunto de dígitos r valuadas y base r requiere al menos b bits para su codificación, donde

$$b = \lceil \log_2 r \rceil = \lfloor \log_2(r - 1) \rfloor + 1$$

Los dígitos decimales en $[0, 9]$ requieren, por tanto, una codificación que al menos tiene cuatro bits de ancho. La representación de *decimal codificado en binario* (BCD, por sus siglas en inglés) se basa en la representación binaria de cuatro bits de dígitos de base 10. Dos dígitos BCD se pueden empaquetar en un byte de ocho bits. Dichas representaciones de *decimal empaquetado* son comunes en algunas calculadoras cuyos circuitos se diseñan para aritmética decimal. Muchas computadoras proporcionan instrucciones especiales para facilitar la aritmética decimal en números BCD empaquetados. Sin embargo, MiniMIPS no tiene tal instrucción.

Desde luego, se pueden usar señales binarias para codificar cualquier conjunto finito de símbolos, no sólo dígitos. Tales codificaciones binarias se usan comúnmente para entrada/salida y transferencias de datos. El American Standard Code for Information Interchange (ASCII), que se muestra en la tabla 6.1, constituye una de dichas convenciones que representan letras mayúsculas y minúsculas, numerales, signos de puntuación y otros símbolos en un byte. Por ejemplo, los códigos ASCII de ocho bits para los diez dígitos decimales son de la forma 0011xxxx, donde la parte “xxxx” es idéntica para el código BCD discutido antes. Los dígitos ASCII toman el doble de espacio que los dígitos BCD y no se usan en unidades aritméticas. Incluso menos compacto que ASCII es el *Unicode* de 16 bits, que puede acomodar símbolos de muchos lenguajes diferentes.

Los números hexadecimales, con $r = 16$, usan dígitos en $[0, 15]$, que se escriben como 0–9, “a” para 10, “b” para 11, . . . y “f” para 15. Ya se ha visto que la notación ensamblador MiniMIPS para números hexa comienza con “0x” seguido por el número hexa. Este tipo de números se pueden ver como una notación abreviada de números binarios, donde cada dígito hexa representa un bloque de cuatro bits.

El uso de valores digitales de 0 a $r - 1$ en base r es sólo una convención. Se podrían usar más de r valores digitales (por ejemplo, valores digitales -2 a 2 en base 4) o usar r valores digitales que no comiencen con 0 (por ejemplo, el conjunto digital $\{-1, 0, 1\}$ en base 3). En la primera instancia, el sistema numérico resultante posee redundancia en el sentido de que algunos números tendrán representaciones múltiples. Los siguientes ejemplos ilustran algunas de las posibilidades.

Ejemplo 9.3: Números ternarios simétricos Si se tuvieran computadoras ternarias en oposición a binarias (con *flip-flops* sustituidos por *flip-flap-flops*), la aritmética en base 3 sería de uso común hoy día. El conjunto digital convencional en base 3 es $\{0, 1, 2\}$. Sin embargo, también se puede usar $\{-1, 0, 1\}$, que es ejemplo de un conjunto digital no convencional. Considere tal sistema numérico *ternario simétrico* de cinco dígitos.

- ¿Cuál es el rango de valores representativos? (Esto es, encontrar las fronteras $máx^-$ y $máx^+$.)
- Represente 3 y -74 como números ternarios simétricos de cinco dígitos.
- Formule un algoritmo para añadir números ternarios simétricos.
- Aplique el algoritmo de suma de la parte c) para encontrar la suma de 35 y -74 .

Solución: El dígito -1 se denotará como $\bar{1}$, para evitar confusión con el símbolo de resta.

- $máx^+ = (1\ 1\ 1\ 1\ 1)_{\text{tres}} = 3^4 + 3^3 + 3^2 + 3 + 1 = 121$; $máx^- = -máx^+ = -121$
- Para expresar un entero como número ternario simétrico, se le debe descomponer en una suma de potencias positivas y negativas de 3. Por tanto, se tiene:

$$35 = 27 + 9 - 1 = (0\ 1\ 1\ 0\ \bar{1})_{\text{tres}} \text{ y } -74 = -81 + 9 - 3 + 1 = (\bar{1}\ 0\ 1\ \bar{1}\ 1)_{\text{tres}}$$

- La suma de dos dígitos en $\{-1, 0, 1\}$ varía de -2 a 2 . En virtud de que $2 = 3 - 1$ y $-2 = -3 + 1$, estos valores se pueden reescribir como dígitos válidos y transferir un acarreo (*carry*) de 1 o -1 (*borrow*, préstamo), que vale 3 o -3 unidades, respectivamente, a la siguiente posición más alta. Cuando se incluye el *carry* ternario entrante, la suma varía de -3 a 3 , que todavía puede manipularse en la misma forma.
- $35 + (-74) = (0\ 1\ 1\ 0\ \bar{1})_{\text{tres}} + (\bar{1}\ 0\ 1\ \bar{1}\ 1)_{\text{tres}} = (0\ \bar{1}\ \bar{1}\ \bar{1}\ 0)_{\text{tres}} = -39$. En el proceso de suma, las posiciones 0 y 1 no producen acarreo, mientras que las posiciones 2 y 3 producen cada una un acarreo de 1.

Ejemplo 9.4: Números con almacenamiento de acarreo En lugar del conjunto convencional de dígitos $\{0, 1\}$ para números binarios, se puede usar en conjunto de dígitos redundante $\{0, 1, 2\}$. El resultado se llama número con *almacenamiento de acarreo*. Considere un sistema numérico con almacenamiento de acarreo de cinco dígitos en el que cada dígito se codifica en dos bits, siendo ambos bits 1 para el valor de dígito 2, o el bit 1 para el valor de dígito 1, y ninguno para 0. Ignore el desbordamiento global y suponga que todos los resultados son representativos en cinco dígitos.

- ¿Cuál es el rango de valores representativos? (Esto es, encuentre las fronteras $máx^-$ y $máx^+$.)
- Represente 22 y 45 como números con almacenamiento de acarreo de cinco dígitos.
- Demuestre que un número binario de cinco bits se puede sumar a un número con almacenamiento de acarreo de cinco dígitos sólo con el uso de operaciones de bits. Esto es, como consecuencia de que no hay propagación de acarreo, tomaría el mismo tiempo hacer la suma incluso si el ancho del operando fuese 64 en lugar de 5.
- Demuestre cómo con el uso de dos pasos de suma libre de acarreo del tipo derivado en la parte c), se puede sumar dos números con almacenamiento de acarreo de cinco dígitos.

Solución

a) $máx^+ = (2\ 2\ 2\ 2\ 2)_{\text{dos}} = 2 \times (2^4 + 2^3 + 2^2 + 2 + 1) = 62$; $máx^- = 0$

- b) Debido a la redundancia, pueden existir múltiples representaciones. Sólo se da una representación para cada entero:

$$22 = (0\ 2\ 1\ 1\ 0)_{\text{dos}} \quad \text{y} \quad 45 = (2\ 1\ 1\ 0\ 1)_{\text{dos}}$$

- c) Esto último se representa mejor gráficamente como en la figura 9.3a. En cada posición existen tres bits: dos del número con almacenamiento de acarreo y uno del número binario. La suma de estos bits es un número de dos de ellos que consiste de un bit suma y un bit acarreo que se conectan esquemáticamente uno a otro mediante una línea. Al sumar tres bits x , y y z para obtener la suma s y el acarreo c es bastante simple y conduce a un circuito lógico simple (un *sumador completo* binario): $s = x \oplus y \oplus z$ y $c = xy \vee yz \vee zx$.

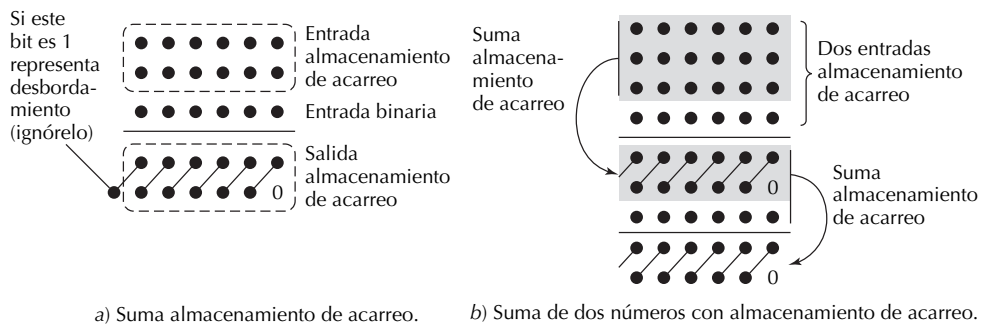


Figura 9.3 Suma de un número binario u otro número con almacenamiento de acarreo, a un número con almacenamiento de acarreo.

- d) Vea el segundo número con almacenamiento de acarreo como dos números binarios. Agregue uno de éstos al primero con almacenamiento de acarreo usando el procedimiento de la parte c) para obtener un número con almacenamiento de acarreo. Luego sume el segundo al número con almacenamiento de acarreo apenas obtenido. De nuevo, la latencia de este proceso de dos pasos es independiente del ancho de los números involucrados. La figura 9.3b contiene una representación gráfica de este proceso.

Además del uso de conjuntos de dígitos no convencionales y redundantes, se han propuesto otras variaciones de representación numérica posicional y han encontrado aplicaciones ocasionales. Por ejemplo, la base r no necesita ser positiva, entera o incluso real. Los *números con base negativa* (conocidos como *negabinarios* para base 2), representaciones con bases irracionales (como $\sqrt{2}$) y sistemas numéricos con base compleja (por ejemplo, $r = 2j$, donde $j = \sqrt{-1}$) son todos factibles [Parh00]. Sin embargo, la discusión de tales sistemas numéricos no compete a este libro.

9.3 Conversión de base numérica

Dado un número x representado en base r , su representación en base R se puede obtener mediante dos formas. Si se quiere realizar aritmética en la nueva base R , se evalúa un polinomio en r cuyos coeficientes son los dígitos x_i . Esto último corresponde a la primera ecuación de la sección 9.1 y se puede realizar más eficientemente con el uso de la regla de Horner, que involucra pasos alternados de multiplicación por r y suma:

$$(x_{k-1}x_{k-2} \cdots x_1x_0)_r = (\cdots((0 + x_{k-1})r + x_{k-2})r + \cdots + x_1)r + x_0$$

Este método es adecuado para la conversión manual de una base arbitraria r a la base 10, dada la relativa facilidad con la que se puede realizar la aritmética en base 10.

Con el propósito de realizar la conversión de base con el uso de aritmética en la antigua base x , se divide repetidamente el número x por la nueva base R , y se conserva el rastro del resto en cada paso. Estos restos corresponden a los dígitos X_i en base R , comenzando desde X_0 . Por ejemplo, el número decimal 19 se convierte a base 3 del modo siguiente:

19 dividido entre 3 produce 6 con resto 1
 6 dividido entre 3 produce 2 con resto 0
 2 dividido entre 3 produce 0 con resto 2

Al leer los restos calculados de arriba abajo, se encuentra $19 = (201)_{\text{tres}}$. Al usar el mismo proceso, se puede convertir 19 a base 5 para obtener $19 = (34)_{\text{cinco}}$.

Ejemplo 9.5: Conversión de binario (o hexa) a decimal Encuentre el equivalente decimal del número binario $(1\ 0\ 1\ 1\ 0\ 1\ 0)_{\text{dos}}$ y destaque cómo se pueden convertir números hexadecimales a decimal.

Solución: Cada 1 en el número binario corresponde a una potencia de 2 y la suma de estas potencias produce el valor decimal equivalente.

$$\begin{array}{cccccccc} 1 & 0 & 1 & & 1 & 0 & 1 & 0 & 1 \\ 128 & + & 32 & + & 16 & + & 4 & + & 1 = 181 \end{array}$$

Se puede llegar a la misma respuesta con el uso de la regla de Horner (lea de izquierda a derecha y de arriba abajo; por ejemplo, comience como $0 \times 2 + 1 \rightarrow 1 \times 2 + 0 \rightarrow 2$, etcétera):

$$\begin{array}{cccccccccccc} 1 & 0 & 1 & & 1 & 0 & 1 & 0 & 1 \\ \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow \\ 0 & 1 & 2 & 5 & 11 & 22 & 45 & 90 & 181 \end{array}$$

La conversión hexadecimal a binario es simple, porque implica sustituir cada dígito hexa con su equivalente binario de cuatro bits; un número hexa de k dígitos se convierte en un número binario de $4k$ bits. Por ende, se puede convertir de hexa a decimal en dos pasos: 1) hexa a binario y 2) binario a decimal.

Ejemplo 9.6: Conversión de decimal a binario (o hexa) Encuentre el equivalente binario de ocho bits de $(157)_{\text{diez}}$ y destaque cómo se deriva el equivalente hexadecimal de un número decimal.

Solución: Para convertir $(157)_{\text{diez}}$ a binario, se necesita dividir entre 2 repetidamente, y anote los restos. La figura 9.4 muestra una justificación para este proceso.

157 dividido entre 2 produce 78 con resto 1
78 dividido entre 2 produce 39 con resto 0
39 dividido entre 2 produce 19 con resto 1
19 dividido entre 2 produce 9 con resto 1
9 dividido entre 2 produce 4 con resto 1
4 dividido entre 2 produce 2 con resto 0
2 dividido entre 2 produce 1 con resto 0
1 dividido entre 2 produce 0 con resto 1



Figura 9.4 Justificación de un paso de la conversión de x a base 2.

Al leer los restos de abajo arriba, se obtiene el equivalente binario deseado $(10011101)_{\text{dos}}$. La figura 9.4 muestra por qué funcionan los pasos del proceso de conversión a base 2. Cuando se obtiene el equivalente binario de un número, su equivalente hexa se forma al tomar grupos de cuatro bits, comenzando con el bit menos significativo, y reescribiéndolos como dígitos hexa. Para el ejemplo, los grupos de cuatro bits son 1001 y 1101, ello conduce al equivalente hexa $(9D)_{\text{hex}}$.

■ **9.4 Enteros con signo**

El conjunto $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$ de enteros también se refiere como *números con signo o dirigidos (enteros)*. La representación más directa de enteros consiste en unir un bit de signo a cualquier codificación deseada de números naturales, lo anterior conduce a representación de *magnitud con signo*. La convención estándar es usar 0 para positivo y 1 para negativo y unir el bit de signo al extremo izquierdo de la magnitud. He aquí algunos ejemplos:

+27 en código binario de magnitud con signo de ocho bits	0 0011011
-27 en código binario de magnitud con signo de ocho bits	1 00110011
-27 en código decimal de dos dígitos con dígitos BCD	1 0010 0111

Otra opción para codificar enteros con signo en el rango $[-N, P]$ es la *representación con signo*. Si se agrega el valor positivo fijo N (el sesgo) a todos los números en el rango deseado, entonces resultan enteros sin signo en el rango $[0, P + N]$. Cualquier método para representar números naturales en $[0, P + N]$ se puede usar para representar los enteros con signo originales en $[-N, P]$. Este tipo de representación sesgada sólo tiene aplicación limitada en la codificación de los exponentes en números de punto flotante (sección 9.6).

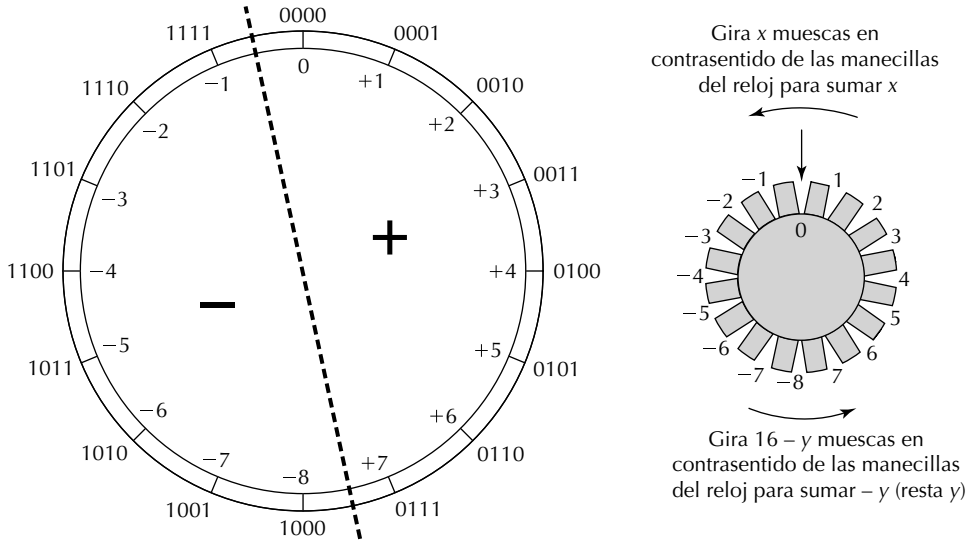


Figura 9.5 Representación esquemática de código en complemento a 2 de cuatro bits para enteros en $[-8, +7]$.

En este contexto, la máquina codificadora más común de enteros con signo constituye la *representación en complemento a 2*. En el formato en complemento a 2 de k bits, un valor negativo $-x$, con $x > 0$, se codifica como el número sin signo $2^k - x$. La figura 9.5 muestra codificaciones de enteros positivos y negativos en el formato a complemento a 2 en cuatro bits. Observe que los enteros positivos del 0 al 7 (o $2^{k-1} - 1$, en general) tienen la codificación binaria estándar, mientras que los valores negativos del -1 al -8 (o -2^{k-1}) se transformaron a valores sin signo al añadirles 16 (o 2^k). El rango de valores representativos en formato en complemento a 2 de k bits es, por tanto $[-2^{k-1}, 2^{k-1} - 1]$.

Cabe destacar dos propiedades importantes de los números en complemento a 2. Primero, el bit más a la izquierda de la representación actúa en el bit de signo (0 para valores positivos, 1 para los negativos). Segundo, el valor representado por un patrón de bit particular se puede derivar sin necesidad de seguir procedimientos diferentes para números negativos y positivos. Simplemente se usa evaluación polinomial o regla de Horner, como se hizo para enteros sin signo, excepto que el bit de signo tiene una importancia negativa. He aquí dos ejemplos:

$$(01011)_{\text{compl a 2}} = (-0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = +11$$

$$(11011)_{\text{compl a 2}} = (-1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = -5$$

La razón para la popularidad de la representación en complemento a 2 se entiende a partir de la figura 9.5. Mientras que en representación de magnitud con signo la suma de números con signos iguales y diferentes implica dos operaciones diferentes (suma vs resta), ambas se realizan en la misma forma con representación en complemento a 2. En otras palabras, la suma de $-y$, que, como siempre, implica rotación en contrasentido a las manecillas del reloj por y muescas, se logra mediante la rotación en contrasentido de las manecillas del reloj por medio de $2^k - y$ muescas. En virtud de que $2^k - y$ constituye la representación en complemento a 2 en k bits de $-y$, la regla común para números tanto negativos como positivos es rotar en contrasentido de las manecillas del reloj por una cantidad igual a la representación del número. Por ende, un sumador binario puede sumar números de cualquier signo.

Ejemplo 9.7: Conversión de complemento a 2 a decimal Encuentre el equivalente decimal del número en complemento a 2 $(1\ 0\ 1\ 1\ 0\ 1\ 0\ 1)_{\text{compl a 2}}$.

Solución: Cada 1 en el número binario corresponde a una potencia de 2 (con el bit de signo considerado como negativo) y la suma de estas potencias produce el valor decimal equivalente.

$$\begin{array}{cccccccc} 1 & 0 & 1 & & 1 & 0 & 1 & 0 & 1 \\ -128 & + & 32 & + & 16 & + & 4 & + & 1 = -75 \end{array}$$

Se puede llegar a la misma respuesta con el uso de la regla de Horner:

$$\begin{array}{cccccccccccc} & 1 & & 0 & & 1 & & 1 & & 0 & & 1 & & 0 & & 1 \\ \times 2 - \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow & \times 2 + \downarrow \\ 0 & -1 & -2 & -3 & -5 & -10 & -19 & -38 & -75 \end{array}$$

Si el número en complemento a 2 se diera con su codificación hexadecimal, esto es, como $(B5)_{\text{hex}}$, se obtendría su equivalente decimal al convertir primero a binario y luego usar el procedimiento anterior.

Observe que, dado que $2^k - (2^k - y) = y$, cambiar el signo de un número en complemento a 2 se realiza mediante la sustracción de 2^k en todos los casos, independientemente de si el número es negativo o positivo. De hecho, $2^k - y$ se calcula como $(2^k - 1) - y + 1$. En virtud de que $2^k - 1$ tiene la representación binaria todos 1, la resta $(2^k - 1) - y$ es simple y consiste de invertir los bits de y . De este modo, como regla, el signo de un número en complemento a 2 se cambia al invertir todos sus bits y luego sumarle 1. La única excepción es -2^{k-1} , cuya negación no es representable en k bits.

Ejemplo 9.8: Cambio de signo para un número en complemento a 2 Como consecuencia de que $y = (1\ 0\ 1\ 1\ 0\ 1\ 0\ 1)_{\text{compl a 2}}$, encuentre la representación en complemento a 2 de $-y$.

Solución: Se necesita invertir todos los bits en la representación de y y luego sumar 1.

$$-y = (0\ 1\ 0\ 0\ 1\ 0\ 1\ 0) + 1 = (0\ 1\ 0\ 0\ 1\ 0\ 1\ 1)_{\text{compl a 2}}$$

Verifique el resultado al convertirlo a decimal:

$$\begin{array}{cccccccc} 0 & 1 & 0 & & 0 & 1 & 0 & 1 & & 1 \\ 64 & & + & & 8 & + & 2 & + & 1 = 75 \end{array}$$

Este último se encuentra en concordancia con $y = -75$, obtenido en el ejemplo 9.7.

La regla anterior para el cambio de signo forma la base de un circuito sumador/restador en complemento a 2 que se muestra en la figura 9.6. Observe que sumar 1 para completar el proceso del cambio de signo se realiza al fijar la entrada de acarreo del sumador a 1 cuando la operación a realizar es la resta; durante la suma, c_{in} se fija a 0. La simplicidad del circuito sumador/restador de la figura 9.6 es una ventaja más de la representación del número en complemento a 2.

También se pueden idear otros sistemas de representación de complemento, pero ninguno está en uso amplio. La elección de cualquier *constante de complementación* M , que es tan grande como $N + P + 1$, permite representar enteros con signo en el rango $[-N, P]$, con los números positivos en $[0, +P]$

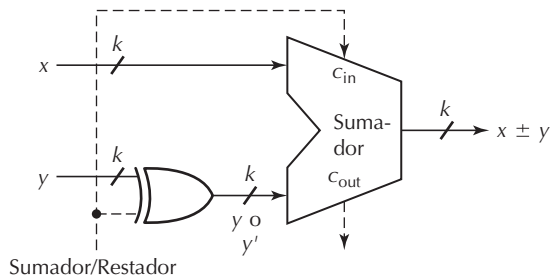


Figura 9.6 Sumador binario usado como sumador/restador en complemento a 2.

correspondiente a los valores sin signo en $[0, P]$ y los números negativos en $[-N, -1]$ representados como valores sin signo en $[M - N, M - 1]$, esto es, al sumar M a valores negativos. A veces, M se reconoce como código alterno para 0 (en realidad, -0). Por ejemplo, el sistema en *complemento a 1* de k bits se basa en $M = 2^k - 1$ e incluye números en el rango simétrico $[-(2^{k-1} - 1), 2^{k-1} - 1]$, donde 0 tiene dos representaciones: la cadena todos 0 y la cadena todos 1.

9.5 Números de punto fijo

Un número de punto fijo consta de una parte *entera o integral* y una parte *fraccional*, con las dos partes separadas por un *punto base* (*punto decimal* en base 10, *punto binario* en base 2, etc.). La posición del punto base casi siempre es implícita y el punto no se muestra explícitamente. Si un número de punto fijo tiene k dígitos enteros y l dígitos fraccionales, su valor se obtiene de la fórmula:

$$x = \sum_{i=-l}^{k-1} x_i r^i = (x_{k-1}x_{k-2} \cdots x_1x_0 \cdot x_{-1}x_{-2} \cdots x_{-l})_r$$

En otras palabras, a los dígitos a la derecha del punto base se les asignan índices negativos y sus pesos son potencias negativas de la base. Por ejemplo:

$$2.375 = (1 \times 2^1) + (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = (10.011)_{\text{dos}}$$

En un sistema numérico de punto fijo en base r y $(k + l)$ dígitos, con k dígitos enteros, se pueden representar los números de 0 a $r^k - r^{-l}$, en incrementos de r^{-l} . El tamaño de paso o resolución r^{-l} con frecuencia se refiere como *ulp*, o *unidad en última posición*. Por ejemplo, en un sistema numérico de punto fijo binario de $(2 + 3)$ bits, se tiene *ulp* = 2^{-3} , y son representables los valores $0 = (00.000)_{\text{dos}}$ al $2^2 - 2^{-3} = 3.875 = (11.11)_{\text{dos}}$. Para el mismo número total $k + l$ de dígitos en un sistema numérico de punto fijo, la k creciente conducirá a un *rango* alargado de números, mientras que l creciente lleva a una mayor *precisión*. Por tanto, existe un intercambio entre rango y precisión.

Los números con signo de punto fijo se representan mediante los mismos métodos discutidos para enteros con signo: magnitud con signo, formato sesgado y métodos de complemento. En particular, para formato en complemento a 2, un valor negativo $-x$ se representa como el valor sin signo $2^k - x$. La figura 9.7 muestra codificaciones de enteros positivos y negativos en el formato en complemento a 2 de punto fijo de $(1 + 3)$ bits. Observe que los valores positivos 0 a $7/8$ (o $2^{k-1} - 2^{-l}$, en general) tienen la codificación binaria estándar, mientras que los valores negativos $-1/8$ a -1 (o -2^{-l} a -2^{k-1} , en general) se transforman en valores sin signo al sumarles 2 (o 2^k , en general).

Las dos propiedades importantes de los números en complemento a 2, ya mencionados en conexión con los enteros, son válidas aquí también; a saber, el bit de la extrema izquierda del número actúa en el

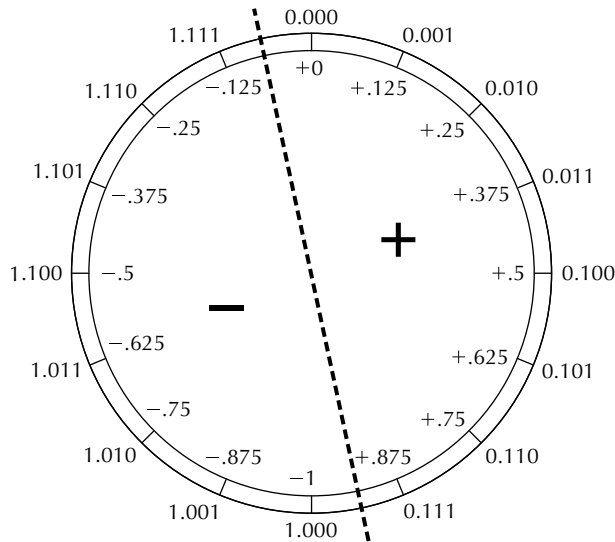


Figura 9.7 Representación esquemática de codificación en complemento a 2 en cuatro bits para números en punto fijo de $(1 + 3)$ bits en el rango $[-1, +7/8]$.

bit de signo, y el valor representado mediante un patrón de bit particular se deriva al considerar el bit de signo con peso negativo. He aquí dos ejemplos:

$$(01.011)_{\text{compl a 2}} = (-0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = +1.375$$

$$(11.011)_{\text{compl a 2}} = (-1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = -0.625$$

El proceso de cambiar el signo de un número es el mismo: invertir todos los bits y sumar 1 a la posición menos significativa (es decir, sumar *ulp*). Por ejemplo, dado $(11.011)_{\text{compl 2}} = -0.625$, su signo cambia al invertir todos los bits en 11.011 y sumar *ulp* para obtener $(00.101)_{\text{compl 2}}$.

La conversión de números de punto fijo de base r a otra base R se hace separadamente para las partes entera y fraccional. Con el propósito de convertir la parte fraccional, de nuevo se puede usar aritmética en la nueva base R o en la antigua base r , la que sea más conveniente. Con aritmética de base R , se evalúa un polinomio en r^{-1} cuyos coeficientes son los dígitos x_i . La forma más simple es ver la parte fraccional como un entero de l dígitos, convertir este entero a base R y dividir el resultado por r^l .

Para realizar la conversión de base con aritmética en la antigua base r , se multiplica repetidamente la fracción y por la nueva base R , anotando y eliminando la parte entera en cada paso. Estas partes enteras corresponden a los dígitos X_{-i} en base R , comenzando desde X_{-1} . Por ejemplo, 0.175 se convierte a base 2 del modo siguiente:

0.175 multiplicado por 2 produce 0.350 con parte entera 0
 0.350 multiplicado por 2 produce 0.700 con parte entera 0
 0.700 multiplicado por 2 produce 0.400 con parte entera 1
 0.400 multiplicado por 2 produce 0.800 con parte entera 0
 0.800 multiplicado por 2 produce 0.600 con parte entera 1
 0.600 multiplicado por 2 produce 0.200 con parte entera 1
 0.200 multiplicado por 2 produce 0.400 con parte entera 0
 0.400 multiplicado por 2 produce 0.800 con parte entera 0

Al leer las partes enteras registradas de arriba abajo, se encuentra $0.175 \cong (.00101100)_{\text{dos}}$. Esta igualdad es aproximada porque el resultado no converge a 0. En general, una fracción en una base puede no tener una representación exacta en otra. En cualquier caso, se efectúa el proceso hasta que se obtenga el número requerido de dígitos en la nueva base. También es posible continuar el proceso después del último dígito de interés, así el resultado se puede redondear.

Ejemplo 9.9: Conversión de binario de punto fijo a decimal Encuentre el equivalente decimal del número binario sin signo $(1101.0101)_{\text{dos}}$. ¿Y si el número dado estuviera en complemento a 2 en lugar de un formato sin signo?

Solución: La parte entera representa $8 + 4 + 1 = 13$. La parte fraccional, cuando se ve como un entero de cuatro bits (multiplicado por 16), representa $4 + 1 = 5$. Por tanto, la parte fraccional es $5/16 = 0.3125$. En consecuencia, $(1101.0101)_{\text{dos}} = (13.3125)_{\text{diez}}$. Para el caso de complemento a 2, se procede igual que arriba: obtener la parte entera $-8 + 4 + 1 = -3$ y la parte fraccional 0.3125 y concluir que $(1101.0101)_{\text{compl 2}} = (-2.6875)_{\text{diez}}$. Así, al reconocer que el número es negativo, se cambia su signo y luego se convierte el número resultante:

$$(1101.0101)_{\text{compl 2}} = -(0010.1011)_{\text{dos}} = -(2 + 11/16) = (-2.6875)_{\text{diez}}$$

Ejemplo 9.10: Conversión de decimal en punto fijo a binario Encuentre los equivalentes en punto fijo binario de $(4 + 4)$ bits de $(3.72)_{\text{diez}}$ y $-(3.72)_{\text{diez}}$.

Solución: El equivalente binario en cuatro bits de 3 es $(0011)_{\text{dos}}$.

0.72 multiplicado por 2 produce 0.44 con parte entera 1
 0.44 multiplicado por 2 produce 0.88 con parte entera 0
 0.88 multiplicado por 2 produce 0.76 con parte entera 1
 0.76 multiplicado por 2 produce 0.52 con parte entera 1
 0.52 multiplicado por 2 produce 0.04 con parte entera 1

Al leer las partes enteras registradas de arriba abajo, se encuentra que $(3.72)_{\text{diez}} = (0011.1100)_{\text{dos}}$. Note que el quinto bit fraccional se obtuvo para redondear la representación al número de punto fijo en $(4 + 4)$ bits más cercano. El mismo objetivo se logró al notar que el resto después del cuarto paso es 0.52, que es mayor que 0.5, y, por tanto, redondearlo. Al obtener sólo cuatro bits fraccionales sin redondeo, la representación binaria habría sido $(0011.1011)_{\text{dos}}$, cuya parte fraccional de $11/16 = 0.6875$ no es una aproximación tan buena a 0.72 como $12/16 = 0.75$. Al usar cambio de signo para el resultado, se encuentra $-(3.72)_{\text{diez}} = (1100.0100)_{\text{compl 2}}$.

9.6 Números de punto flotante

Los enteros en un rango preescrito se pueden representar exactamente para procesamiento automático, pero la mayoría de los números reales se deben aproximar dentro del ancho de palabra finito de la máquina. Algunos números reales se representan como, o aproximarse mediante, números de

punto fijo de $(k + l)$ bits, como se vio en la sección 9.5. Un problema con la representación en punto fijo es que no son muy buenas para tratar con números muy grandes o pequeños al mismo tiempo. Considere los dos números de punto fijo de $(8 + 8)$ bits que se muestran abajo:

$$\begin{aligned}x &= (0000\ 0000 . 0000\ 1001)_{\text{dos}} && \text{número pequeño} \\y &= (1001\ 0000 . 0000\ 0000)_{\text{dos}} && \text{número grande}\end{aligned}$$

El error relativo de representación debido a truncado o redondeo de dígitos más allá de la $-8a$ posición es bastante significativa para x , pero menos severa para y . Asimismo, ni y^2 ni y/x es representativa en este formato numérico.

Por ende, la representación en punto fijo parece ser inadecuada para aplicaciones que manejan valores numéricos en amplio rango, desde muy pequeño hasta en extremo grande, ya que necesitaría una palabra muy ancha para acomodar al mismo tiempo los requisitos de rango y de precisión. Los *números en punto flotante* constituyen el modo primario de la aritmética en tales situaciones. Un valor en punto flotante consta de un número con magnitud con signo y punto fijo y un factor de escala acompañante. Después de muchos años de experimentación con diferentes formatos de punto flotante y tener que sobrellevar las inconsistencias e incompatibilidades resultantes, la informática aceptó el formato estándar propuesto por el Institute of Electrical and Electronics Engineers (IEEE, Instituto de Ingenieros Eléctricos y Electrónicos) y adoptado por organizaciones similares nacionales e internacionales, incluida la American National Standards Institute (ANSI). Por lo tanto, la discusión de los números y aritmética de punto flotante se formula sólo en términos de este formato. Otros formatos diferirán en sus parámetros y detalles de representación, pero las negociaciones básicas y algoritmos permanecerán iguales.

Un número en punto flotante en el estándar ANSI/IEEE tiene tres componentes: signo \pm , exponente e y significando s , que representan el valor $\pm 2^e s$. El *exponente* es un entero con signo representado en formato sesgado (se le suma un sesgo fijo para hacerlo un número sin signo). El *significando* constituye un número en punto fijo en el rango $[1, 2)$. Como consecuencia de que la representación binaria del significando siempre comienza con “1.”, este 1 fijo se omite (*hidden*) y sólo la parte fraccional del significando se representa de manera explícita.

La tabla 9.1 y la figura 9.8 muestran los detalles de los formatos ANSI/IEEE en punto flotante corto (32 bits) y largo (64 bits). El formato corto tiene rango y precisión adecuados para las aplicaciones más comunes (magnitudes que varían de 1.2×10^{-38} a 3.4×10^{38}). El formato largo se usa para cálculos precisos o éstos incluyen variaciones extremas en magnitud (desde 2.2×10^{-308} hasta 1.8×10^{308}). Desde luego que en estos formatos, como ya se explicó, el cero no tiene una representación propia, pues el significando siempre es distinto de cero. Para remediar este problema, y ser capaz de representar otros valores especiales, los códigos de exponente más pequeño y más grande (todos 0 y todos 1 en el campo exponente sesgado) no se usan para números ordinarios. Una palabra toda 0 (0 en los campos signo, exponente y significando) representa $+0$; de igual modo -0 y $\pm\infty$ tienen representaciones especiales, como lo tienen cualquier valor indeterminado, conocido como “no número” (NaN). Otros detalles de este estándar no se contemplan en este libro; en particular, aquí no se explican los números desnormalizados (*denormales*, para abreviar), que se incluyen en la tabla 9.1.

Cuando una operación aritmética produce un resultado que no es representativo en el formato que se usa, el resultado debe redondearse a algún valor semejante. El estándar ANSI/IEEE prescribe cuatro opciones de redondeo. El modo de redondeo por defecto es *redondeo al par más cercano*: elegir el valor representativo más cercano y, para el caso de empate, elegir el valor con su bit menos

■ **TABLA 9.1** Algunas características de los formatos punto flotante estándar ANSI/IEEE.

Característica	Sencillo/corto	Doble/largo
Ancho de palabra en bits	32	64
Bits significando	23 + 1 omitido	52 + 1 omitido
Rango significando	$[1, 2 - 2^{-23}]$	$[1, 2 - 2^{-52}]$
Bits exponente	8	11
Sesgo exponente	127	1023
Cero (± 0)	$e + \text{sesgo} = 0, f = 0$	$e + \text{sesgo} = 0, f = 0$
Número desnormalizado	$e + \text{sesgo} = 0, f \neq 0$ representa $\pm 0 \cdot f \times 2^{-126}$	$e + \text{sesgo} = 0, f \neq 0$ representa $\pm 0 \cdot f \times 2^{-1022}$
Infinito ($\pm \infty$)	$e + \text{sesgo} = 255, f = 0$	$e + \text{sesgo} = 2047, f = 0$
No número (NaN)	$e + \text{sesgo} = 255, f \neq 0$	$e + \text{sesgo} = 2047, f \neq 0$
Número ordinario	$e + \text{sesgo} \in [1, 254]$ $e \in [-126, 127]$ representa $1 \cdot f \times 2^e$	$e + \text{sesgo} \in [1, 2046]$ $e \in [-1022, 1023]$ representa $1 \cdot f \times 2^e$
mín	$2^{-126} \cong 1.2 \times 10^{-38}$	$2^{-1022} \cong 2.2 \times 10^{-308}$
máx	$\cong 2^{128} \cong 3.4 \times 10^{38}$	$\cong 2^{1024} \cong 1.8 \times 10^{308}$

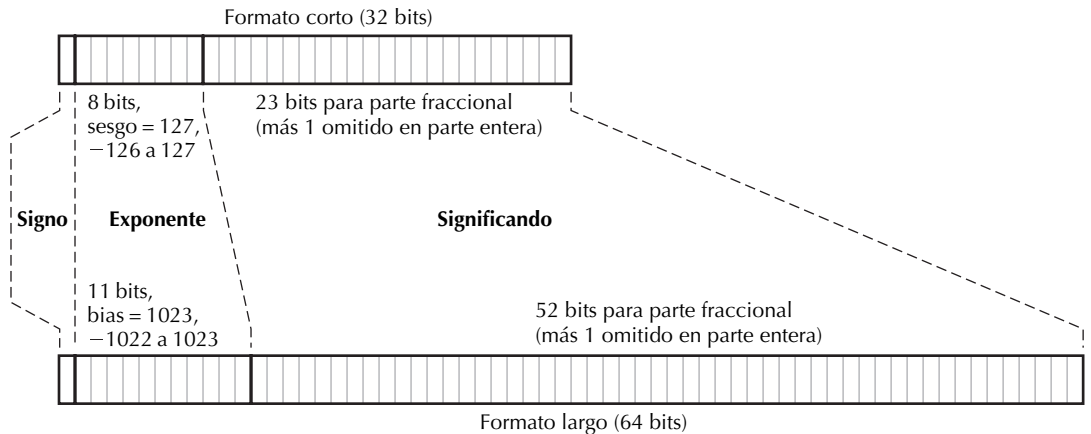


Figura 9.8 Los dos formatos punto flotante estándar ANSI/IEEE.

significativo 0. También existen tres modos de *redondeo dirigido*: *redondeo hacia $+\infty$* (elegir el siguiente valor más alto), *redondeo hacia $-\infty$* (elegir el siguiente valor más bajo) y *redondeo hacia 0* (elegir el valor más cercano que es menor que el valor más próximo en magnitud). Con la opción de redondear al más cercano, el error de redondeo máximo es 0.5 *ulp*, mientras que con los esquemas de redondeo dirigido, el error puede ser hasta de 1 *ulp*. En la sección 12.1 se discute con más detalle el redondeo de valores de punto flotante.

PROBLEMAS

9.1 Sistemas numéricos posicionales de base fija

Para cada uno de los siguientes sistemas numéricos convencionales de base r , y el uso de los valores de dígito del 0 al $r - 1$, determine: $máx^-$, $máx^+$, el número b de bits necesario para codificar un dígito, el número K de dígitos necesarios para representar el equivalente de todos los enteros binarios sin signo de 32 bits y la eficiencia de representación en el último caso.

- $r = 2, k = 24$
- $r = 5, k = 10$
- $r = 10, k = 7$
- $r = 12, k = 7$
- $r = 16, k = 6$

9.2 Sistemas numéricos posicionales de base fija

Pruebe los enunciados de las partes *a*) a la *c*):

- Un entero binario sin signo es una potencia de 2 ssi la AND lógica de bits de x y $x - 1$ es 0.
- Un entero sin signo en base 3 es par ssi la suma de todos sus dígitos es par.
- Un entero binario sin signo $(x_{k-1}x_{k-2} \cdots x_1x_0)_{\text{dos}}$ es divisible entre 3 ssi $\sum_{\text{par } i} x_i - \sum_{\text{impar } i} x_i$ es un múltiplo de 3.
- Generalice los enunciados de las partes *b*) y *c*) para obtener reglas de divisibilidad de enteros en base r por $r - 1$ y $r + 1$.

9.3 Desbordamiento en operaciones aritméticas

Argumente si ocurrirá desbordamiento cuando cada una de las expresiones aritméticas se evalúa dentro del marco de los cinco sistemas numéricos definido en el problema 9.1 (20 casos en conjunto). Todos los operandos están dados en base 10.

- 3000×4000
- $2^{24} - 2^{22}$
- $9000000 + 500000 + 400000$
- $1^2 + 2^2 + 3^2 + \cdots + 2000^2$

9.4 Conjuntos de dígitos no convencionales

Considere un sistema numérico simétrico de punto fijo en base 3 con k dígitos enteros y l fraccionales, y use el

conjunto de dígitos $[-1, 1]$. La versión entera de esta representación se discutió en el ejemplo 9.3.

- Determine el rango de los números representados como función de k y l .
- ¿Cuál es la eficiencia representacional relativa a la representación binaria, como consecuencia de que cada dígito en base 3 necesita una codificación de dos bits?
- Diseñe un procedimiento de hardware simple y rápido para convertir de la representación anterior a una representación en base 3 con el uso del conjunto de dígitos redundante $[0, 3]$.
- ¿Cuál es la eficiencia representacional de la representación redundante en la parte *c*)?

9.5 Números con almacenamiento de acarreo

En el ejemplo 9.4:

- Proporcione todas las posibles representaciones alternas para la parte *b*).
- Identifique los números, si existen, que tengan representaciones únicas.
- ¿Cuál entero tiene el mayor número posible de representaciones distintas?

9.6 Números con almacenamiento de acarreo

La figura 9.3 se puede interpretar como una forma de comprimir tres números binarios en dos números binarios que tienen la misma suma, y el mismo ancho, ignorando la posibilidad de desbordamiento. De igual modo, la figura 9.3b muestra la compresión de cuatro números binarios, primero a tres y luego a dos números. Un circuito de hardware que convierte tres números binarios a dos con la misma suma se denomina *sumador con almacenamiento de acarreo*.

- Dibuje un diagrama similar a la figura 9.3b que represente la compresión de seis números binarios a dos que tengan la misma suma y ancho.
- Convierta su solución a la parte *a*) a un diagrama de hardware, use bloques de tres entradas y dos salidas para representar los sumadores con almacenamiento de acarreo.
- Demuestre que el diagrama de hardware de la parte *b*) tiene una latencia igual a la de tres sumadores con almacenamiento de acarreo, de otro modo presente

una implementación en hardware distinta con tal latencia para comprimir seis números binarios a dos de éstos.

9.7 Números negabinarios

Los números negabinarios usan el conjunto de dígitos $\{0, 1\}$ en base $r = -2$. El valor de un número negabinario se evalúa en la misma forma que un número binario, pero los términos que contienen potencias impares de la base son negativos. Por tanto, los números positivos y negativos se representan sin necesidad de un bit de signo separado o un esquema de complementación.

- ¿Cuál es el rango de valores representables en representaciones negabinarias de nueve y diez bits?
- Dado un número negabinario, ¿cómo se determina su signo?
- Diseñe un procedimiento para convertir un número negabinario positivo a un número binario sin signo convencional.

9.8 Números decimales comprimidos

Una forma de representar números decimales en memoria consiste en empaquetar dos dígitos BCD en un byte. Esta representación es poco práctica, pues se usa un byte que puede codificar 256 valores para representar los pares de dígitos del 00 al 99. Una forma de mejorar la eficiencia es comprimir tres dígitos BCD en diez bits.

- Diseñe una codificación adecuada para esta compresión. *Sugerencia:* Sean los tres dígitos BCD $x_3x_2x_1x_0$, $y_3y_2y_1y_0$ y $z_3z_2z_1z_0$. Sea $WX_2X_1x_0Y_2Y_1y_0Z_2Z_1z_0$ la codificación de diez bits. En otras palabras, se usan los últimos bits menos significativos (LSB, por sus siglas en inglés) de los tres dígitos y los restantes nueve bits (tres de cada dígito) se codifican en siete bits. Sea $W = 0$ que codifica el caso $x_3 = y_3 = z_3 = 0$. En este caso, los dígitos restantes se copian en la nueva representación. Use $X_2X_1 = 00, 01, 10$ para codificar el caso en que sólo uno de los valores x_3, y_3 o z_3 sea 1. Observe que cuando el bit más significativo de un dígito BCD es 1, el dígito está completamente especificado por su LSB y no se necesita ninguna otra información. Finalmente, use $X_2X_1 = 11$ para todos los otros casos.

- Diseñe un circuito para convertir tres dígitos BCD en la representación comprimida en diez bits.
- Diseñe un circuito para descomprimir el código de diez bits para recuperar los tres dígitos BCD originales.
- Sugiera una codificación similar para comprimir dos dígitos BCD en siete bits.
- Diseñe la compresión requerida y los circuitos de descompresión para la codificación de la parte d).

9.9 Conversión de base numérica

Convierta cada uno de los números siguientes de su base indicada a representación en base 10.

- Números en base 2: 1011, 1011, 0010, 1011 0010 1111 0001
- Números en base 3: 1021, 1021 2210, 1021 2210 2100 1020
- Números en base 8: 534, 534 607, 534 607 126 470
- Números en base 12: 7a4, 7a4 539, 7a4 593 1b0
- Números en base 16: 8e, 8e3a, 8e3a 51c0

9.10 Números en punto fijo

Un sistema numérico binario de punto fijo tiene 1 bit entero y 15 bits fraccionales.

- ¿Cuál es el rango de números representado, si supone un formato sin signo?
- ¿Cuál es el rango de números representado, si supone formato en complemento a 2?
- Represente las fracciones decimales 0.75 y 0.3 en el formato de la parte a).
- Represente las fracciones decimales -0.75 y -0.3 en el formato de la parte b).

9.11 Conversión de base numérica

Convierta cada uno de los números siguientes de su base indicada a representaciones en base 2 y base 16.

- Números en base 3: 1021, 1021 2210, 1021 2210 2100 1020
- Números en base 5: 302, 302 423, 302 423 140
- Números en base 8: 354, 534 607, 534 607 126 470
- Números en base 10: 12, 5 655, 2 550 276, 76 545 336, 3 726 755
- Números en base 12: 9a5, b0a, ba95, a55a1, baabaa

9.12 Conversión de base numérica

Convierta cada uno de los siguientes números en punto fijo de su base indicada a representación en base 10.

- a) Números en base 2: 10.11, 1011.0010, 1011 0010.1111 001
- b) Números en base 3: 10.21, 1021.2210, 1021 2210.2100 1020
- c) Números en base 8: 53.4, 534.607, 534 607.126 470
- d) Números en base 12: 7a.4, 7a4.539, 7a4 593.1b0
- e) Números en base 16: 8.e, 8e.3a, 8e3a.51c0

9.13 Conversión de base numérica

Convierta cada uno de los siguientes números en punto fijo de su base indicada a representación en base 2 y base 16.

- a) Números en base 3: 10.21, 1021.2210, 1021 2210.2100 1020
- b) Números en base 5: 30.2, 302.423, 302 423.140
- c) Números en base 8: 53.4, 534.607, 534 607. 126 470
- d) Números en base 10: 1.2, 56.55, 2 550.276, 76 545.336, 3 726.755
- e) Números en base 12: 9a.5, b.0a, ba.95, a55a1, baa. baa

9.14 Formato de complemento a dos

- a) Codifique cada uno de los siguientes números decimales en formato complemento a 2 de 16 bits con bits fraccionales 0, 2, 4 u 8: 6.4, 33.675, 123.45.

- b) Verifique lo correcto de sus conversiones en la parte a) mediante la aplicación de la fórmula de Horner a los números en complemento a 2 resultantes para derivar los equivalentes decimales.
- c) Repita la parte a) para los siguientes números decimales: -6.4, 33.675, -123.45
- d) Repita la parte b) para los resultados de la parte c).

9.15 Números en punto flotante

Considere las entradas proporcionadas por *mín* y *máx* en la tabla 9.1. Muestre cómo se derivan dichos valores y explique por qué los valores *máx* en las dos columnas se especifican como casi iguales a una potencia de 2.

9.16 Números en punto flotante

Muestre la representación de los siguientes números decimales en formatos en punto flotante corto y largo ANSI/IEEE. Cuando el número no es exactamente representable, use la regla del redondeo al par más cercano.

- a) 12.125
- b) 555.5
- c) 333.3
- d) -6.25
- e) -1024.0
- f) -33.2

REFERENCIAS Y LECTURAS SUGERIDAS

- [Gold91] Goldberg, D., "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *ACM Computing Surveys*, vol. 23, núm. 1, pp. 5-48, marzo de 1991.
- [IEEE85] *ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic*, disponible de IEEE Press.
- [Knut97] Knuth, D. E., *The Art of Computer Programming*, vol. 2: *Seminumerical Algorithms*, Addison-Wesley, 3a. ed., 1997.

- [Kore93] Koren, I., *Computer Arithmetic Algorithms*, Prentice Hall, 1993.
- [Parh00] Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.
- [Parh02] Parhami, B., “Number Representation and Computer Arithmetic”, en *Encyclopedia of Information Systems*, Academic Press, 2002, vol. 3, pp. 317-333.
- [Swar90] Swartzlander, E. E., Jr, *Computer Arithmetic*, vols. I y II, IEEE Computer Society Press, 1990.

■ CAPÍTULO 10

SUMADORES Y ALU SIMPLES

“Escribí este libro y compilé en él todo lo que es necesario para la computación, evitando tanto la verborrea aburrida como la brevedad confusa.”

Ghiyath al-Din Jamshid al-Kashi, La llave de la computación (Miftah al-Hisabi), 1427

“En consecuencia, parece que, cualquiera que sea el número de dígitos que el motor analítico sea capaz de retener, si se requiere hacer todos los cálculos con k veces dicho número de dígitos, entonces lo puede ejecutar el mismo motor, pero en la cantidad de tiempo igual a k^2 veces que el anterior.”

Charles Babbage, Pasajes de la vida de un filósofo, 1864

TEMAS DEL CAPÍTULO

- 10.1** Sumadores simples
- 10.2** Redes de propagación de acarreo
- 10.3** Conteo e incremento
- 10.4** Diseño de sumadores rápidos
- 10.5** Operaciones lógicas y de corrimiento
- 10.6** ALU multifunción

La suma representa la operación aritmética más importante en las computadoras digitales. Incluso las computadoras incrustadas más simples tienen un sumador, mientras que los multiplicadores y divisores se encuentran sólo en los microprocesadores de alto rendimiento. En este capítulo se comienza por considerar el diseño de sumadores de un solo bit (medio sumadores y sumadores completos) y se muestra cómo tales bloques constructores se pueden poner en cascada para construir sumadores con acarreo en cascada. Luego se procede con el diseño de sumadores más rápidos construidos mediante anticipación de acarreo, el método de predicción de acarreo más usado. Otros temas cubiertos incluyen contadores, operaciones de corrimiento y lógicas, y ALU multifunción.

■ 10.1 Sumadores simples

En este capítulo se cubren sólo suma y resta entera binaria. Los números en punto fijo que están en el mismo formato se pueden sumar o restar como enteros al ignorar el punto base implícito. La suma con punto flotante se tratará en el capítulo 12.

Cuando dos bits se suman, la suma es un valor en el rango $[0, 2]$ que se puede representar mediante un *bit suma* y un *bit acarreo* (*carry*). El circuito que puede calcular los bits suma y *carry* se denomina

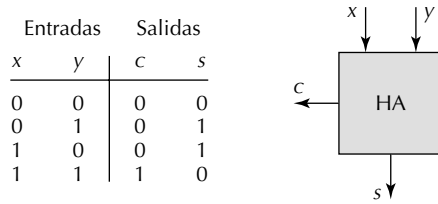


Figura 10.1 Tabla de verdad y diagrama esquemático para un medio sumador (HA) binario.

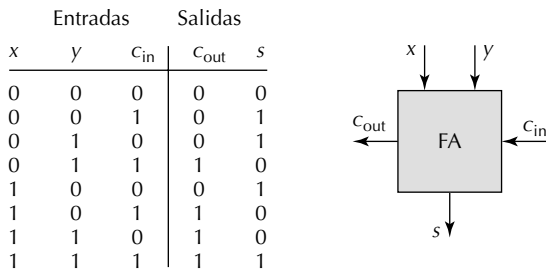


Figura 10.2 Tabla de verdad y diagrama esquemático para un sumador completo (FA) binario.

medio sumador (HA) binario, y en la figura 10.1 se muestran su tabla de verdad y su representación simbólica. La salida de acarreo es la AND lógica de las dos entradas, mientras que la salida suma es la OR exclusiva (XOR) de las entradas. Al añadir una entrada de *carry* a un medio sumador, se obtiene un *sumador completo* (FA) binario, cuya tabla de verdad y diagrama esquemático se muestran en la figura 10.2. En la figura 10.3 se muestran muchas implementaciones de un sumador completo.

Un sumador completo, conectado a un *flip-flop* para retener el bit *carry* de un ciclo al siguiente, funciona como un *sumador de bits en serie*. Las entradas de un sumador de bits en serie se proporcionan en sincronía con una señal de reloj, un bit de cada operando por ciclo de reloj, comenzando con los bits menos significativos (LSB, por sus siglas en inglés). Por ciclo de reloj se produce un bit de la salida, y el acarreo de un ciclo se retiene y usa como entrada en el ciclo siguiente. Asimismo, un *sumador de acarreo en cascada* despliega este comportamiento secuencial en el espacio, con el uso de una cascada de k sumadores completos para sumar dos números de k bits (figura 10.4).

El diseño en acarreo en cascada de la figura 10.4 se convierte en un sumador de base r si cada sumador completo binario se sustituye por un sumador completo de base r que acepta dos dígitos en base r (cada uno codificado en binario) y una señal de entrada de acarreo (*carry-in*), que produce un dígito suma en base r y una señal de salida de acarreo (*carry-out*). Cuando se conocen todas las señales de acarreo intermedias c_i , los bits/dígitos suma se calculan con facilidad. Por esta razón, las discusiones de diseño de sumador se enfocan en cómo se pueden derivar todos los acarreos intermedios, según los operandos de entrada y c_{in} . Puesto que las señales de acarreo siempre son binarias y su propagación se puede hacer independientemente de la base r , como se discute en la sección 10.2, a partir de este punto con frecuencia no se tratará con bases distintas de 2.

Observe que cualquier diseño de sumador binario se puede convertir a un sumador/restador en complemento a 2 a través del esquema que se muestra en la figura 9.6. Por esta razón no se discutirá la resta como una operación separada.

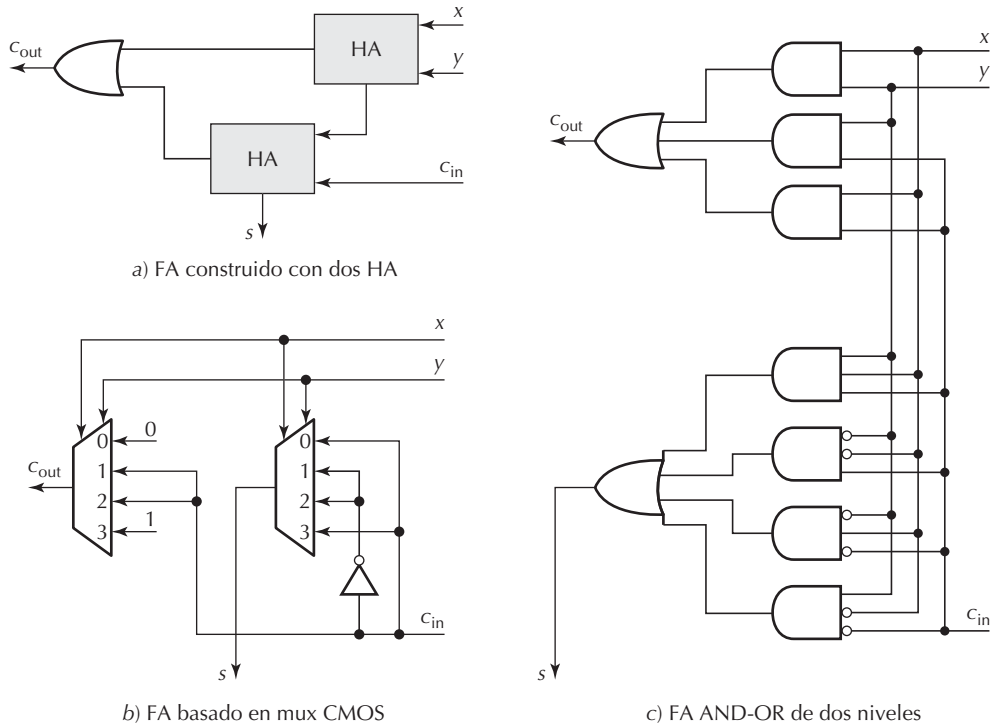


Figura 10.3 Sumador completo implementado con dos medios sumadores, por medio de dos multiplexores de cuatro entradas y como una red de compuertas de dos niveles.

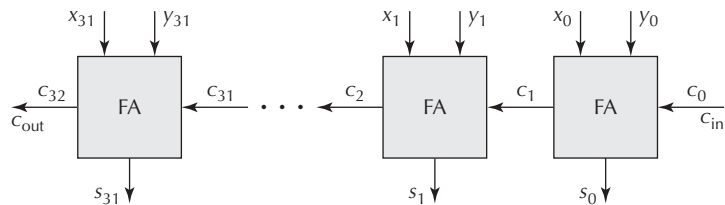


Figura 10.4 Sumador binario con acarreo en cascada, con entradas y salidas de 32 bits.

10.2 Redes de propagación de acarreo

Cuando se suman dos números, los acarreos se generan en ciertas posiciones de dígitos. Para la suma decimal se trata de posiciones en las que la suma de los dígitos operando es 10 o más. En suma binaria, la generación de acarreo requiere que ambos bits operando sean 1. Se define a la señal binaria auxiliar g_i como 1 para las posiciones en las que se genera un acarreo y 0 para cualquier otra parte. Para la suma binaria, $g_i = x_i y_i$; esto es, g_i es la AND lógica de los bits operando x_i y y_i . De igual modo, existen posiciones de dígito en las que se propaga un acarreo entrante. Para suma decimal, éstas son las posiciones en las que la suma de los dígitos operando es igual a 9; un acarreo de entrada hace 10 la suma de posición, lo que conduce a un acarreo de salida desde dicha posición. Desde luego, si

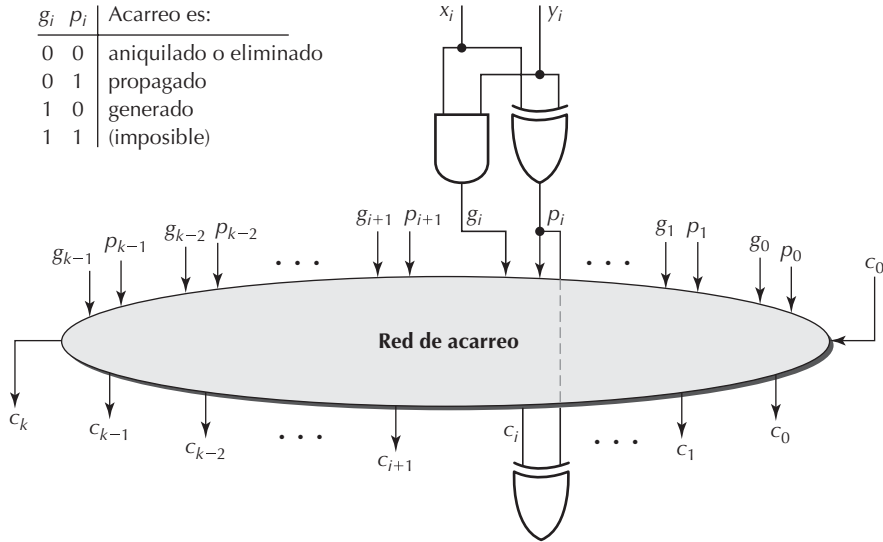


Figura 10.5 La parte principal de un sumador es la red de acarreo. El resto es sólo un conjunto de compuertas para producir las señales g y p y los bits suma.

no hay acarreo de entrada no habrá una salida de acarreo para tales posiciones. En suma binaria, la propagación de acarreo requiere que un bit operando sea 0 y el otro sea 1. La señal binaria auxiliar p_i , derivada como $p_i = x_i \oplus y_i$ para suma binaria, se define como 1 ssi la posición del dígito i propaga un acarreo entrante.

Ahora, según el acarreo entrante c_0 de un sumador y las señales auxiliares g_i y p_i para todas las posiciones de dígito k , se pueden derivar los acarreos intermedios c_i y el acarreo saliente c_k independientemente de los valores digitales de entrada. En un sumador binario, el bit suma en la posición i se deriva como

$$s_i = x_i \oplus y_i \oplus c_i = p_i \oplus c_i$$

En la figura 10.5 se muestra la estructura general de un sumador binario. Las variaciones en la red de acarreo resultan en muchos diseños que difieren en sus costos de implementación, rapidez operativa, consumo de energía, etcétera.

El sumador con acarreo en cascada de la figura 10.4 es muy simple y fácilmente expandible a cualquier ancho deseado. Sin embargo, más bien es lento porque los acarreos se pueden propagar a través de todo el ancho del sumador. Esto ocurre, por ejemplo, cuando se suman los dos números de ocho bits 10101011 y 01010101. Cada sumador completo requiere cierto tiempo para generar su salida de acarreo con base en su entrada de acarreo y los bits operandos en dicha posición. Hacer en cascada k de tales unidades implica k veces tanto retardo de señal en el peor caso. Esta cantidad lineal de tiempo se vuelve inaceptable para palabras anchas (por decir, 32 o 64 bits) o en computadoras de alto rendimiento, aunque puede ser aceptable en un sistema incrustado que se dedica a una sola tarea y no se espera que sea rápido.

Con el propósito de ver un sumador con acarreo en cascada en el marco general de la figura 10.5, observe que la red de acarreo de un sumador con acarreo en cascada se basa en la recurrencia:

$$c_{i+1} = g_i \vee p_i c_i$$

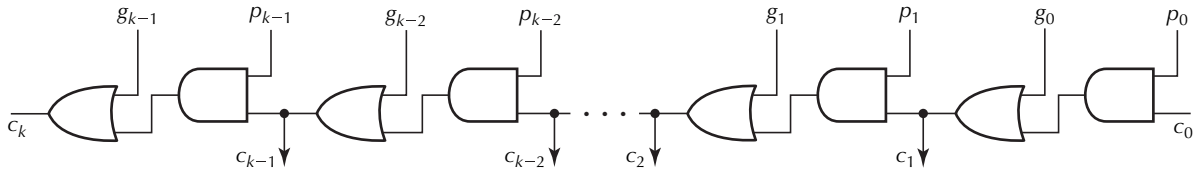


Figura 10.6 Red de propagación de acarreo de un sumador con acarreo en cascada.

Esta recurrencia destaca que un acarreo irá a la posición $i + 1$ si se genera en la posición i o si un acarreo que entra en la posición i se propaga por la posición i . Esta observación conduce a la figura 10.6 como la red de acarreo de un sumador con acarreo en cascada. La latencia lineal de un sumador con acarreo en cascada ($2k$ niveles de compuerta para la red de acarreo, más unos cuantos más para derivar las señales auxiliares y producir los bits suma) es evidente a partir de la figura 10.6.

Ejemplo 10.1: Variaciones en diseño de sumador Se dice que, en la posición de dígito i , ocurre una transferencia si un acarreo se genera o propaga. Para sumadores binarios, la señal de transferencia auxiliar $t_i = g_i \vee p_i$ se puede derivar por una compuerta OR, dado que $g_i \vee p_i = x_i y_i + (x_i \oplus y_i) = x_i \vee y_i$. Con frecuencia, una compuerta OR es más rápida que una compuerta XOR, de modo que t_i se puede producir más rápido que p_i .

- Demuestre que la recurrencia de acarreo $c_{i+1} = g_i \vee p_i c_i$ sigue siendo válida si se sustituye p_i con t_i .
- ¿Cómo afecta al diseño de una red de acarreo el cambio de la parte a)?
- ¿En qué otras formas el cambio de la parte a) afecta el diseño de un sumador binario?

Solución

- Se demuestra que las dos expresiones $g_i \vee p_i c_i$ y $g_i \vee t_i c_i$ son equivalentes al convertir una en la otra: $g_i \vee p_i c_i = g_i \vee g_i c_i \vee p_i c_i = g_i \vee (g_i \vee p_i) c_i = g_i \vee t_i c_i$. Observe que, en el primer paso de la conversión, la inclusión del término adicional $g_i c_i$ se justifica por $g_i \vee g_i c_i = g_i (1 \vee c_i) = g_i$; en otras palabras, el término $g_i c_i$ es redundante.
- En virtud de que cambiar p_i a t_i no afecta la relación entre c_{i+1} y c_i , nada cambiará en la red de acarreo si se le proporciona t_i en lugar de p_i . La pequeña diferencia en rapidez entre t_i y p_i lleva a una producción más rápida de las señales de acarreo.
- Se necesita incluir k compuertas OR de dos entradas adicionales para producir las señales t_i . Todavía se requieren las señales p_i para producir los bits suma $s_i = p_i \oplus c_i$ una vez que todos los acarreos se conozcan. No obstante, el sumador será más rápido global, porque las señales p_i se derivan con el funcionamiento de la red de acarreo, que se acota para tardar más.

Existen algunas formas de acelerar la propagación de acarreo, ello conduce a suma más rápida. Un método, que es conceptualmente bastante simple, consiste en proporcionar rutas de salto en una red con acarreo en cascada. Por ejemplo, una red de acarreo de 32 bits se puede dividir en ocho secciones de cuatro bits, con una compuerta AND de cinco entradas que permita a los acarreos entrantes de la posición $4j$ ir directamente al final de la sección en caso de que $p_{4j} = p_{4j+1} = p_{4j+2} = p_{4j+3} = 1$. En la figura 10.7 se bosqueja una sección de cuatro bits de la red de acarreo que abarca las posiciones de bit de la $4j$ a la $4j + 3$. Observe que la latencia de tal sumador con trayectorias de salto de cuatro bits todavía es lineal en k , aunque es mucho menor que la de un sumador simple con acarreo en cascada. La propagación más rápida de acarreos a través de las trayectorias de salto se puede parecer a

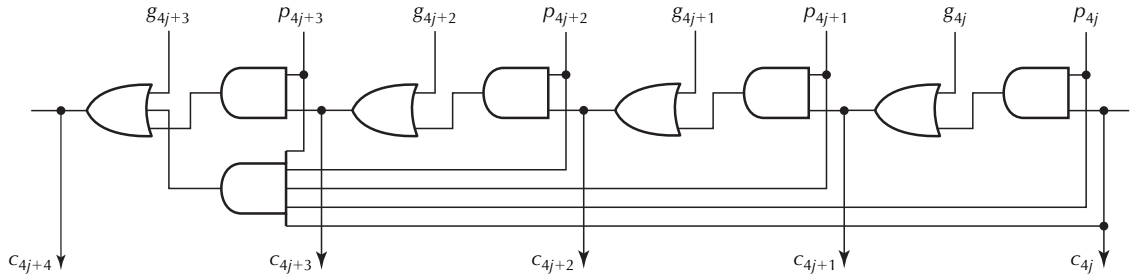


Figura 10.7 Sección de cuatro bits de una red con acarreo en cascada con trayectorias de salto.

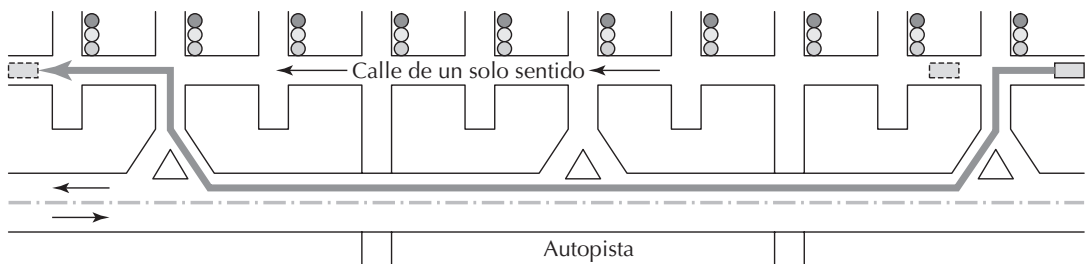


Figura 10.8 Analogía de automovilismo para propagación de acarreo en sumadores con trayectorias de salto. Tomar la autopista permite al conductor que quiere recorrer una larga distancia y evitar retardos excesivos en muchos semáforos.

los conductores que reducen sus tiempos de viaje al usar una autopista cercana siempre que el destino deseado esté muy cerca (figura 10.8).

Ejemplo 10.2: Ecuación de acarreo para sumadores con salto Se vio que los sumadores con acarreo en cascada implementan la recurrencia de acarreo $c_{i+1} = g_i \vee p_i c_i$. ¿Cuál es la ecuación correspondiente para el sumador con acarreo en cascada con trayectorias de salto de cuatro bits, que se muestra en la figura 10.7?

Solución: Es evidente de la figura 10.7 que la ecuación de acarreo permanece igual para cualquier posición cuyo índice i no es múltiplo de 4. La ecuación para el acarreo entrante en la posición $i = 4j + 4$ se convierte en $c_{4j+4} = g_{4j+3} \vee p_{4j+3}c_{4j+3} \vee p_{4j+3}p_{4j+2}p_{4j+1}p_{4j}c_{4j}$. Esta ecuación muestra que existen tres formas (no mutuamente excluyentes) en las que un acarreo puede entrar a la posición $4j + 4$: al generarse en la posición $4j + 3$, a través de la propagación de un acarreo que entra a la posición $4j + 3$, o que el acarreo en la posición $4j$ pase a lo largo de la trayectoria de salto.

10.3 Conteo e incremento

Antes de discutir algunas de las formas comunes en que se acelera el proceso de la suma en las computadoras modernas, considere un caso especial de suma, el de que uno de los dos operandos sea una constante. Si un registro se inicializa a un valor x y luego repetidamente se le suma una constante a , se obtendrá la secuencia de valores $x, x + a, x + 2a, x + 3a, \dots$. Este proceso se conoce como conteo por a . La figura 10.9 muestra una implementación en hardware de este proceso con el uso de un sumador

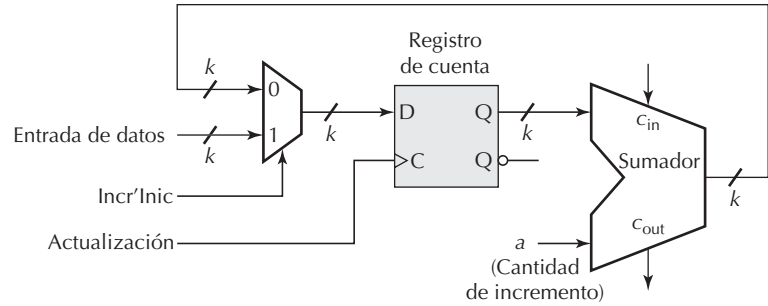


Figura 10.9 Diagrama esquemático de un contador síncrono inicializable.

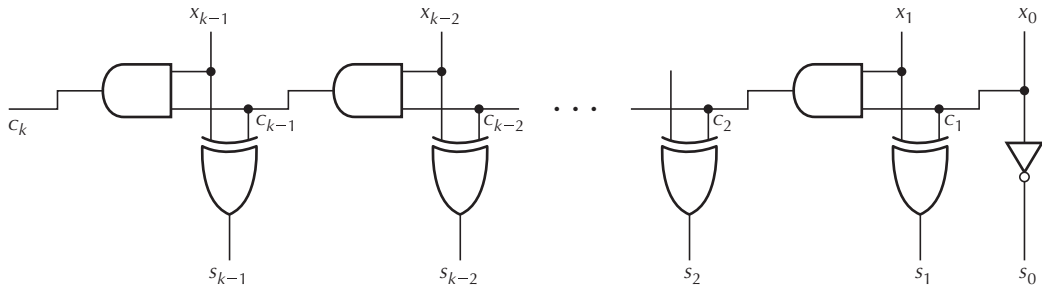


Figura 10.10 Red de propagación de acarreo y lógica de suma para un incrementador.

cuya entrada inferior está conectada permanentemente a la constante a . Este contador se puede actualizar de dos modos diferentes: el *incremento* provoca que el contador pase al siguiente valor en la secuencia anterior, y la *inicialización* causa que un valor de datos de entrada se almacene en el registro.

El caso especial de $a = 1$ corresponde al estándar *contador arriba* cuya secuencia es $x, x + 1, x + 2, x + 3, \dots$, mientras que $a = -1$ produce un *contador hacia abajo*, que procede en el orden $x, x - 1, x - 2, x - 3, \dots$; un *contador arriba/abajo* puede contar hacia arriba o abajo, según el valor de una señal de control de dirección. Si en el proceso de contar hacia abajo se pasa por 0, el contador se fija al valor negativo en complemento a 2 adecuado. Los contadores arriba y abajo pueden desbordarse cuando la cuenta se vuelve demasiado grande o pequeña para representarse dentro del formato de representación numérica utilizada. En el caso de contadores arriba sin signo, el desbordamiento se indica mediante la salida de acarreo del sumador que se postula. En otros casos, el desbordamiento del contador se detecta de la misma forma que en los sumadores (sección 10.6).

Ahora se enfocará la atención en un contador hacia arriba con $a = 1$. En este caso, en lugar de conectar la constante 1 a la entrada inferior del sumador, como en la figura 10.9, se puede fijar $c_{in} = 1$ y usar 0 como la entrada de sumador inferior. Entonces, el sumador se convierte en un *incrementador* cuyo diseño es más simple que un sumador ordinario. Para ver por qué, observe que al sumar $y = 0$ a x , se tienen las señales de generación $g_i = x_i y_i = 0$ y las señales de propagación $p_i = x_i \oplus y_i = x_i$. Por tanto, al referirse a la red de propagación de acarreo de la figura 10.6, se ve que se pueden eliminar todas las compuertas OR, así como la compuerta AND de la extrema derecha, ello conduce a la red de acarreo simplificada de la figura 10.10. Más adelante se verá que los métodos usados para acelerar

la propagación de acarreo en los sumadores se pueden adoptar también para diseñar incrementadores más rápidos.

El contador de programa de una máquina es un ejemplo de un contador hacia arriba que se incrementa para apuntar a la instrucción siguiente conforme se ejecuta la instrucción actual. El incremento PC sucede más rápido en el ciclo de ejecución de instrucción, así que hay mucho tiempo para su terminación, lo anterior implica que puede no requerirse un incrementador superrápido. Para el caso de MiniMIPS, el PC aumenta no por 1 sino por 4; sin embargo, el incremento por cualquier potencia de 2, tal como 2^k , es lo mismo que ignorar los h bits menos significativos y añadir 1 a la parte restante.

10.4 Diseño de sumadores rápidos

Es posible diseñar una diversidad de sumadores rápidos que requieran tiempo logarítmico en lugar de lineal. En otras palabras, el retardo de tales sumadores rápidos crece como el logaritmo de k . Los más conocidos y ampliamente usados de tales sumadores son los *sumadores con anticipación de acarreo*, cuyo diseño se discute en esta sección.

La idea básica en la suma con anticipación de acarreo es formar los acarreos intermedios requeridos directamente de las entradas g_i , p_i y c_{in} hacia la red de acarreo, en lugar de a partir de los acarreos previos, como se hace en los sumadores con acarreo en cascada. Por ejemplo, el acarreo c_3 del sumador de la figura 10.4, que previamente se expresó en términos de c_2 con el uso de la recurrencia de acarreo

$$c_3 = g_2 \vee p_2 c_2$$

se puede derivar directamente de las entradas con base en la expresión lógica:

$$c_3 = g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_0$$

Esta expresión se obtiene al desarrollar la recurrencia original; esto es, al sustituir c_2 con su expresión equivalente en términos de c_1 y luego expresar c_1 en términos de c_0 . De hecho, se podría escribir esta expresión directamente con base en la siguiente explicación intuitiva. Un acarreo hacia la posición 3 debe haberse generado en algún punto a la derecha de la posición de bit 3 y propagarse desde ahí hacia la posición de bit 3. Cada término al lado derecho de la ecuación anterior cubre una de las cuatro posibilidades.

Teóricamente, se pueden desarrollar todas las ecuaciones de acarreo y obtener cada uno de los acarreos como una expresión AND-OR de dos niveles. Sin embargo, la expresión completamente desarrollada se volvería muy larga para un sumador más ancho que requiere que, por decir, se deriven c_{31} o c_{52} . Existen diversas *redes de acarreo con anticipación de acarreo* que sistematizan la derivación precedente para todos los acarreos intermedios en paralelo y hacen el cálculo eficiente al compartir partes de los circuitos requeridos siempre que sea posible. Varios diseños ofrecen negociaciones en rapidez, costo, área de chip VLSI y consumo de energía. Información acerca del diseño de redes con anticipación de acarreo y otros tipos de sumadores rápidos se puede encontrar en libros de aritmética computacional [Parh00].

Aquí, sólo se presenta un ejemplo de una red con anticipación de acarreo. Los bloques constructores de esta red constan del *operador acarreo*, que combina las señales generar y propagar para dos bloques adyacentes $[i + 1, j]$ y $[h, i]$ de posiciones de dígito en las señales respectivas para el bloque combinado más ancho $[h, j]$. En otras palabras,

$$[i + 1, j] \phi [h, i] = [h, j]$$

donde ϕ designa el operador acarreo y $[a, b]$ se pone para $(g_{[a,b]}, p_{[a,b]})$, que representa el par de señales generar y propagar para el bloque que se extiende desde la posición de dígito a hacia la posición de

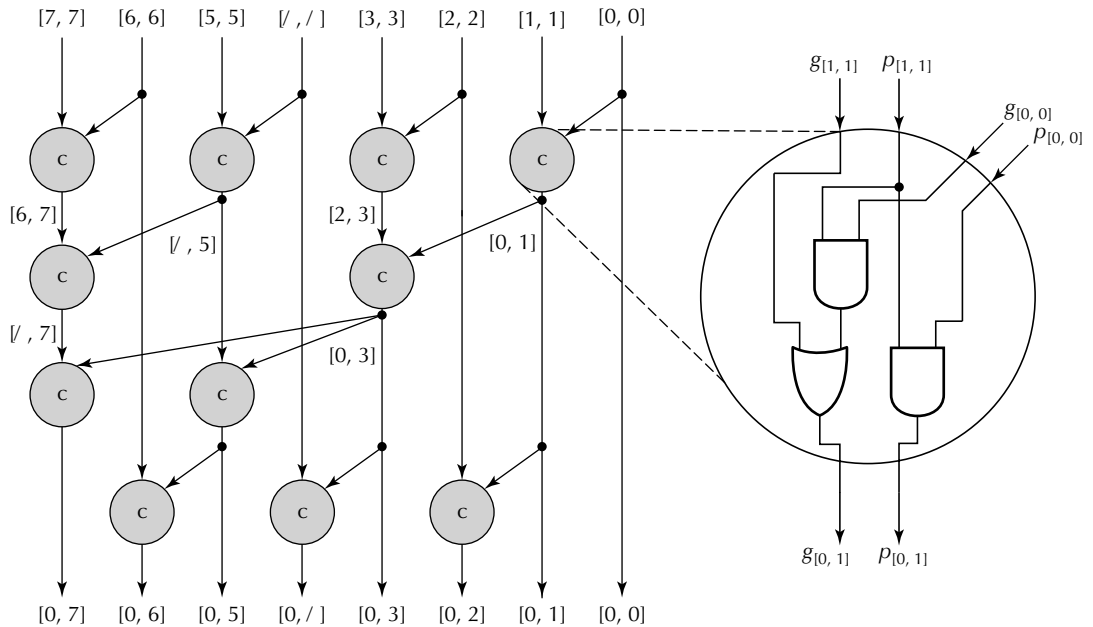


Figura 10.11 Red con anticipación de acarreo Brent-Kung para un sumador de ocho dígitos, junto con detalles de uno de los bloques operadores de acarreo.

dígito b . Puesto que el problema de determinar todos los acarreos c_{i+1} es el mismo que calcular las señales generar acumulativas $g_{[0,i]}$, se puede usar una red construida de bloques operador ϕ , como la que se muestra en la figura 10.11, para derivar todos los acarreos en paralelo. Si una señal c_{in} se requiere para el sumador, se puede acomodar como la señal generar g_{-1} de una posición adicional a la derecha; en este caso se necesitaría una red de acarreo de $(k + 1)$ bits para un sumador de k bits.

Para entender mejor la red de acarreo de la figura 10.11 y ser capaz de analizar su costo y latencia en general, observe su estructura recursiva [Bren82]. La hilera superior de operadores de acarreo combinan señales g y p de nivel bit en señales g y p para bloques de dos bits. Las últimas señales corresponden a las señales generar y propagar para una suma en base 4, donde cada dígito en base 4 consta de dos bits en el número binario original. Las hileras restantes de operadores de acarreo, que básicamente consiste de todos los otros acarreos (los numerados pares) en la suma original en base 2. Todo lo que queda es para la hilera inferior de operadores de acarreo para proporcionar los acarreos con número impar perdidos.

En la figura 10.12 se acentúa esta estructura recursiva al mostrar cómo la red Brent-Kung de ocho bits de la figura 10.11 se compone de una red de cuatro entradas del mismo tipo más dos hileras de operadores de acarreo, una hilera en lo alto y otra en el fondo. Esto conduce a una latencia que corresponde a casi $2 \log_2 k$ operadores de acarreo (dos hileras, por $\log_2 k$ niveles de recursión) y un costo aproximado de $2k$ bloques operador (casi k bloques en dos hileras en el comienzo, luego $k/2$ bloques, $k/4$ bloques, etc.). Los valores exactos son ligeramente menores: $2 \log_2 k - 2$ niveles para latencia y $2k - \log_2 k - 2$ bloques para costo.

En lugar de combinar las señales de acarreo auxiliares dos a la vez, lo que conduce a $2 \log_2 k$ niveles, uno puede hacer combinaciones de cuatro vías para operación más rápida. Dentro de un grupo de cuatro bits, abarcar las posiciones de bit 0 a 3, el grupo de señales generar y propagar se derivan como:

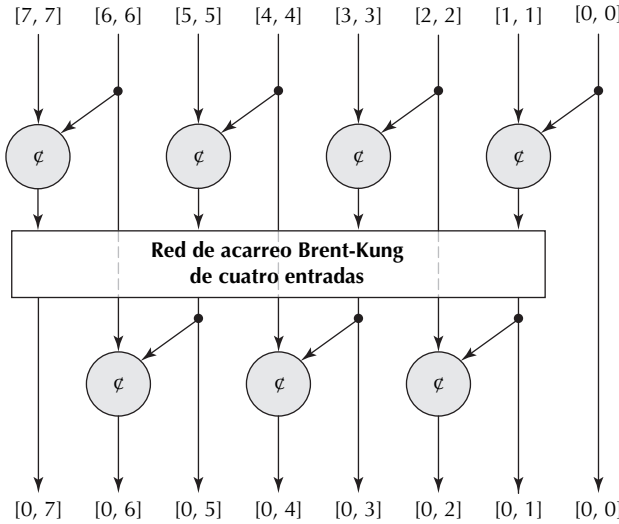


Figura 10.12 Red con anticipación de acarreo Brent-Kung para un sumador de ocho dígitos, donde sólo se muestran sus hilas superior e inferior de operadores de acarreo.

$$g_{[0,3]} = g_3 \vee p_3 g_2 \vee p_3 p_2 g_1 \vee p_3 p_2 p_1 g_0$$

$$p_{[0,3]} = p_3 p_2 p_1 p_0$$

Cuando se conocen las señales g y p por grupos de 4 bits, el mismo proceso se repite para los $k/4$ pares de señales resultantes. Esto lleva a la derivación de los acarreos intermedios c_4, c_8, c_{12}, c_{16} , etc.; esto es, uno en cada cuatro posiciones. El problema restante es determinar los acarreos intermedios dentro de grupos de cuatro bits. Esto se puede hacer con anticipación completa. Por ejemplo, los acarreos c_1, c_2 y c_3 en el grupo de la extrema derecha se deriva como

$$c_1 = g_0 \vee p_0 c_0$$

$$c_2 = g_1 \vee p_1 g_0 \vee p_1 p_0 c_0$$

$$c_3 = g_2 \vee p_2 g_1 \vee p_2 p_1 g_0 \vee p_2 p_1 p_0 c_0$$

El circuito resultante será similar en estructura a la red de acarreo Brent-Kung, excepto que la mitad superior de hilas consistirá de bloques de producción de grupos g y p y la mitad inferior serán los bloques de producción de acarreos intermedios apenas definidos. La figura 10.13 muestra los diseños de estos dos tipos de bloques.

Un importante método para el diseño de un sumador rápido, que con frecuencia complementa el esquema de anticipación de acarreo, es la *selección de acarreo*. En la aplicación más simple del método de selección de acarreo, se construye un sumador de k bits a partir de un sumador de $(k/2)$ bits en la mitad inferior, dos sumadores de $(k/2)$ bits en la mitad superior (que forman dos versiones de los $k/2$ bits suma superiores con $c_{k/2} = 0$ y $c_{k/2} = 1$) y un multiplexor para escoger el conjunto de valores correcto una vez que se conozca $c_{k/2}$. Un diseño híbrido, en el que algunos de los acarreos (por decir, c_8, c_{16} y c_{24} en un sumador de 32 bits) se derivan a través de bloques con anticipación de acarreo y se utilizan para seleccionar una de dos versiones de sumadores de bits, éstas son producidas para bloques de 8 bits concurrentemente con la operación de la red de acarreo es muy popular en las unidades aritméticas modernas. La figura 10.14 muestra una parte de tal sumador con selección de acarreo, que elige la versión correcta de la suma para las posiciones de bit de la a a la b antes de conocer el acarreo intermedio c_a .

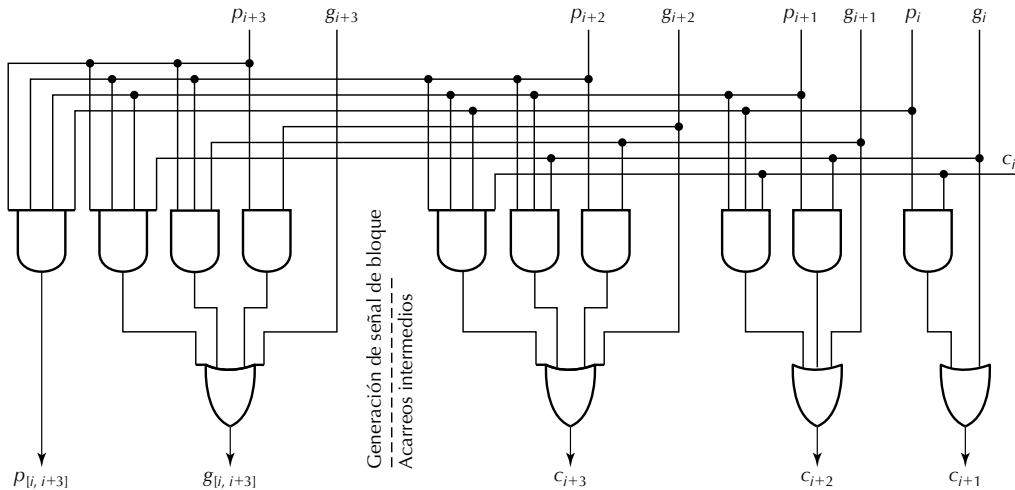


Figura 10.13 Bloques necesarios en el diseño de sumadores con anticipación de acarreo con agrupamiento de bits de cuatro vías.

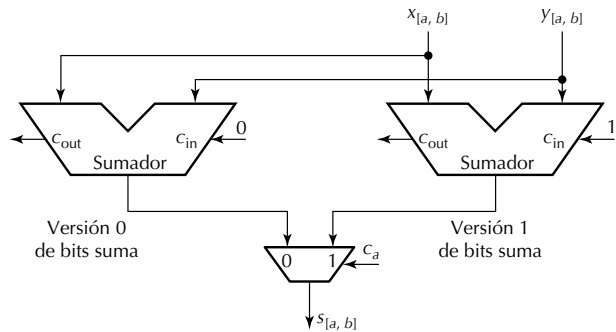


Figura 10.14 Principio de suma con selección de acarreo.

10.5 Operaciones lógicas y de corrimiento

Implementar operaciones lógicas dentro de una ALU es muy simple, dado que el i -ésimo bit de los dos operandos se combinan para producir el i -ésimo bit del resultado. Tales operaciones de bits se pueden implementar mediante un arreglo de compuertas, como se muestra en la figura 1.5. Por ejemplo, *and*, *or*, *nor*, *xor*, y otras instrucciones lógicas de MiniMIPS se implementan con facilidad de esta forma. Controlar cuál de estas operaciones realiza la ALU es algo que se discutirá en la sección 10.6.

El corrimiento involucra un reordenamiento de bits dentro de una palabra. Por ejemplo, cuando una palabra se corre lógicamente a la derecha por cuatro bits, el bit i de la palabra de entrada constituirá el bit $i - 4$ de la palabra de salida, y los cuatro bits más significativos de la palabra de salida se llenan con 0. Suponga que se quiere correr una palabra de 32 bits a la derecha o izquierda (señal de control *right/left* = derecha/izquierda) por una cantidad dada como un número binario de cinco bits (0 a 31 bits). Conceptualmente, esto último se puede lograr con el uso de un multiplexor 64 a 1 con entradas de 32 bits (figura 10.15). Sin embargo, prácticamente, el circuito resultante es demasiado complejo, en particular cuando también se incluyen otros tipos de corrimiento (aritmético, cíclico). Los corredores

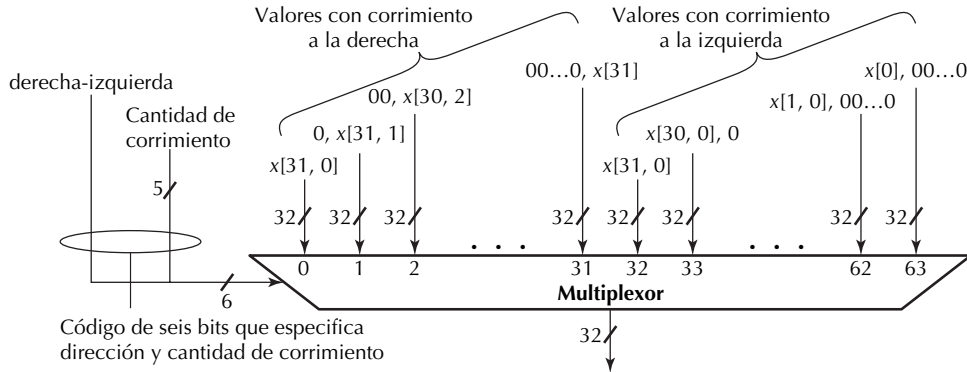


Figura 10.15 Unidad de corrimiento lógico con base en multiplexor.

(*shifters*) prácticos usan una implementación multinivel. Después de discutir la noción de corrimientos aritméticos, se describirá tal implementación.

Corrimiento a la izquierda h bits un número x de k bits sin signo multiplica su valor por 2^h , siempre que dicho $2^h x$ sea representable en k bits. Lo anterior sucede porque cada 0 adicionado a la derecha de un número binario duplica su valor (en la misma forma en que los números decimales se multiplican por 10 con cada 0 que se añade a su extremo derecho). De manera sorprendente, esta observación también se aplica a números en complemento a 2 de k bits. Dado que el signo de un número no debe cambiar cuando se multiplica por 2^h , el número en complemento a 2 debe tener $h + 1$ bits idénticos en su extremo izquierdo si ha de multiplicarse por 2^h de esta forma. Esto asegurará que, luego de que h bits se han descartado como resultado del corrimiento izquierdo, el valor de bit en la posición de signo no cambiará.

El corrimiento derecho lógico afecta de modo diferente a los números en complemento a 2 y sin signo. Un número x sin signo se divide entre 2^h cuando se corre a la derecha por h bits. Esto es comparable con mover el punto decimal a la izquierda una posición para dividir un número decimal entre 10. Sin embargo, este método de dividir entre 2^h no funciona para números en complemento a dos negativos; tales números se vuelven positivos cuando se insertan 0 desde la izquierda en el curso de corrimiento derecho lógico. La división propia de un número en complemento a 2 entre 2^h requiere que los bits que entran de la izquierda sean los mismos que el bit signo (0 para números positivos y 1 para negativos). Este proceso se conoce como corrimiento derecho aritmético. En virtud de que dividir números por potencias de 2 es muy útil, y muy eficiente cuando se hace a través de corrimiento, la mayoría de las computadoras tienen provisiones para dos tipos de corrimiento derecho: corrimiento derecho lógico, que ve el número como una cadena de bits cuyos bits se reposicionarán vía corrimiento, y corrimiento derecho aritmético, cuyo propósito es dividir el valor numérico del operando por una potencia de 2.

MiniMIPS tiene dos instrucciones de corrimiento aritmético: *shift right arithmetic* y *shift right arithmetic variable*. Esos se definen igual a los corrimientos derechos lógicos, excepto que, durante el corrimiento ocurre extensión de signo, como se discutió anteriormente.

```
sra  $t0, $s1, 2    # fija $t0 en ($s1) corrido derecho por 2
srav $t0, $s1, $s0  # fija $t0 en ($s1) corrido derecho por ($s0)
```

La figura 10.16 muestra las representaciones en máquina de estas dos instrucciones.

Ahora está listo para discutir el diseño de un *shifter* para corrimientos tanto lógicos como aritméticos. Primero, considere el caso de corrimientos de un solo bit. Se quiere diseñar un circuito que pueda



Figura 10.16 Las dos instrucciones de corrimiento aritmético de MiniMIPS.

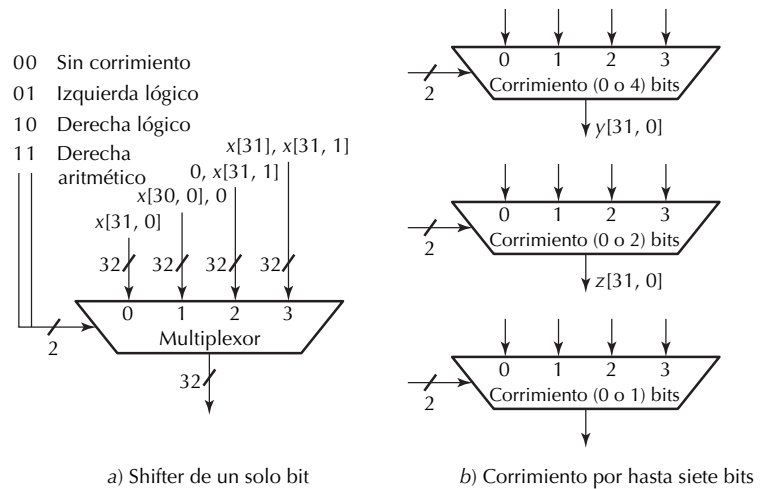
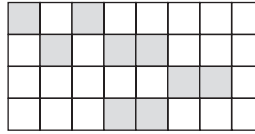


Figura 10.17 Corrimiento multietapas en un shifter de barril.

realizar corrimiento izquierdo lógico de 1 bit, corrimiento derecho lógico de 1 bit o corrimiento derecho aritmético de 1 bit con base en las señales de control proporcionadas. Es conveniente considerar el caso de no corrimiento como una cuarta posibilidad y usar la codificación que se muestra en la figura 10.17a para distinguir los cuatro casos. Si el operando de entrada es $x[31, 0]$, la salida debe ser $x[31, 0]$ para no corrimiento, $x[30, 0]$ adicionado con 0 para corrimiento izquierdo lógico, $x[31, 1]$ precedido por 0 para corrimiento derecho lógico y $x[31, 1]$ precedido por $x[31]$ (es decir, una copia del bit signo) para corrimiento derecho aritmético. Por tanto, se puede usar un multiplexor de cuatro entradas para realizar cualquiera de estos corrimientos, como se muestra en la figura 10.17a.

Los corrimientos multibit se pueden realizar en muchas etapas con el uso de réplicas del circuito que se muestra en la figura 10.17a para cada etapa. Por ejemplo, suponga que se van a realizar corrimientos lógico y aritmético con cantidades de corrimiento entre 0 y 7, proporcionados como un número binario de tres bits. Las tres etapas que se muestran en la figura 10.17b realizan los corrimientos deseados al efectuar primero un corrimiento de cuatro bits, si se requiere, con base en el bit más significativo de la cantidad de corrimiento. Esto último convierte la entrada $x[31, 0]$ a $y[31, 0]$. El valor intermedio y se sujeta entonces a corrimiento de dos bits si el bit medio de la cantidad de corrimiento es 1, lo que

Bloque de 32 pxeles (4×8)
de imagen blanco y negro:



Representación como palabra de 32 bits:

Renglón 0	Renglón 1	Renglón 2	Renglón 3
1010 0000	0101 1000	0000 0110	0001 0111

Equivalente hexa: 0xa0a80617

Figura 10.18 Bloque de 4×8 de una imagen en blanco y negro que se representa como una palabra de 32 bits.

conduce al resultado z . Finalmente, z se corre por 1 bit si el bit menos significativo de la cantidad de corrimiento es 1. El diseño multietapa de la figura 10.17b se denomina *corredor de barril (barrel shifter)*. Es sencillo agregar corrimientos cíclicos, o rotaciones, a los diseños de la figura 10.17.

Las instrucciones *logical* y *shift* tienen muchas aplicaciones. Por ejemplo, se pueden usar para identificar y manipular campos o bits individuales dentro de palabras. Suponga que se quiere aislar los bits del 10 al 15 en una palabra de 32 bits. Una forma de hacerlo es operar AND la palabra con la “máscara”

0000 0000 0000 0000 1111 1100 0000 0000

que tiene 1 en las posiciones de bit de interés y 0 en cualquier otra parte, y luego se corre el resultado a la derecha (lógicamente) por diez bits para llevar los bits de interés al extremo derecho de la palabra. En este punto, la palabra resultante tendrá un valor numérico en el rango $[0, 63]$ dependiendo de los contenidos de la palabra original en las posiciones de bit 10 al 15.

Como segundo ejemplo, considere una palabra de 32 bits como representación de un bloque de 4×8 de una imagen blanco y negro, con 1 representando un pixel oscuro y 0 un pixel blanco (figura 10.18). Los valores de pixel se identifican individualmente al realizar de manera alternada corrimientos izquierdos de 1 bit y verificar el signo del número. Una prueba inicial identifica el primer pixel (un número negativo significa 1). Después de un corrimiento izquierdo, el segundo pixel se identifica mediante la prueba del signo. Esto último se continúa hasta conocer todos los valores de pixel.

10.6 ALU multifunción

Ahora se puede juntar todo lo visto en este capítulo y presentar el diseño de una ALU multifunción que realice operaciones suma/resta, lógicas y de corrimiento. Sólo se considerarán las operaciones aritméticas/lógicas necesarias para ejecutar las instrucciones de la tabla 5.1; esto es, suma, resta, AND, OR, XOR, NOR. La estructura global de la ALU se muestra en la figura 10.19. Consta de tres subunidades para corrimiento, suma/resta y operaciones lógicas. La salida de una de dichas subunidades, o el MSB (bit más significativo) de la salida sumador (el bit de signo), extendida a 0 para una palabra completa de 32 bits, se puede elegir como la salida de ALU al postular las señales de control “clase de función” (*function class*) del multiplexor de cuatro entradas. En el resto de esta sección se describe cada una de las tres subunidades en la ALU.

Primero, se verá el sumador. Éste es el sumador en complemento a 2 descrito en la figura 9.6. Lo que se añadió aquí es un circuito NOR de 32 entradas cuya salida es 1 ssi la salida del sumador es 0 (detector todos 0) y una compuerta XOR que sirve para detectar el desbordamiento. El desbordamiento en la aritmética de complemento a 2 sólo ocurre cuando los operandos de entrada son del mismo signo

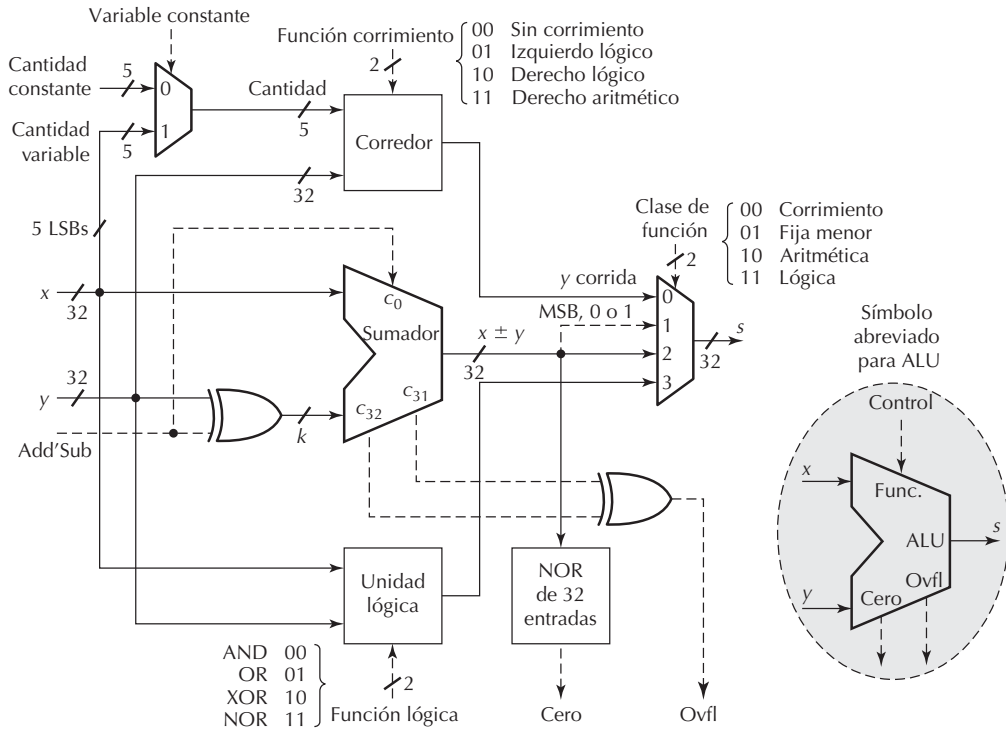


Figura 10.19 ALU multifunción con ocho señales de control (dos para clase de función, uno para aritmética, tres corrimientos, dos lógicos) que especifican la operación.

y la salida de sumador es de signo opuesto. Por ende, al denotar los signos de las entradas y salidas por x_{31} , y_{31} y s_{31} , la siguiente expresión para la señal desbordamiento se deriva fácilmente:

$$\text{Ovfl} = x_{31}y_{31}s'_{31} \vee x'_{31}y'_{31}s_{31}$$

Una formulación equivalente, cuya comprobación se deja como ejercicio, es aquella en que un desbordamiento ocurre cuando el siguiente al último acarreo del sumador (c_{31}) es diferente de su salida de acarreo (c_{32}). Lo anterior es la forma en que se implementa la detección de desbordamiento en la figura 10.19.

A continuación, considere de nuevo el shifter de barril (figura 10.17). Puesto que MiniMIPS tiene dos clases de instrucción shift (proporcionadas en el campo “sh” de la instrucción) o en una variable shift amount (en un registro designado), se usa un multiplexor para dirigir la cantidad adecuada al shifter. Note que, cuando la cantidad de corrimiento está dada en un registro, sólo son relevantes los últimos 5 bits menos significativos de la cantidad (¿por qué?).

La unidad lógica realiza una de las cuatro operaciones lógicas de bits de AND, OR, XOR, NOR. De modo que el diseño de la unidad lógica consiste de cuatro arreglos de compuertas que calculan dichas funciones lógicas y un multiplexor de cuatro entradas que permite elegir el conjunto de resultados que avanza a la salida de la unidad.

Finalmente, proporcionar el MSB de la salida del sumador como una de las entradas al multiplexor de cuatro entradas requiere alguna explicación. El bit de signo es extendido 0 para formar una palabra completa de 32 bits cuyo valor es 0 si el bit de signo es 0 y 1 si el bit de signo es 1. Esto permite usar

la ALU para implementar la instrucción `slt` de MiniMIPS, que requiere que 1 o 0 se almacene en un registro dependiendo de si $x < y$ o no lo es. Esta condición se puede verificar al calcular $x - y$ y ver el bit signo del resultado: si el bit signo es 1, entonces $x < y$ y se sostiene y se debe almacenar 1 en el registro designado; de otro modo, se debe almacenar 0. Esto es exactamente lo que la ALU proporciona como su salida cuando sus señales “clase de función” que controlan al multiplexor de salida se fijan a 01.

PROBLEMAS

10.1 Diseño de medio sumadores

Con una compuerta AND y una compuerta XOR se puede realizar un medio sumador. Demuestre que también se puede realizar con:

- Cuatro compuertas NAND de dos entradas y un inversor (o sin inversor si la señal c_{out} se produce en forma invertida).
- Tres compuertas NOR de dos entradas y dos inversores (o sin inversor si las entradas x y y están disponibles en formas tanto verdadera como complementada).

10.2 Sumadores con acarreo en cascada

- Suponga que cada bloque FA de la figura 10.4 se implementa como en la figura 10.3c y que las compuertas de tres entradas son más lentas que las compuertas de dos entradas. Dibuje una línea que representa la trayectoria crítica en la figura 10.4.
- Ahora, suponga que los FA se implementan como en la figura 10.3b y derive el retardo de la trayectoria crítica de la figura 10.4 en términos de mux y retardos de inversor.

10.3 Contador con decremento unitario

En la sección 10.3 se mostró que insertar un valor con incremento de 1 a través de la entrada de acarreo del sumador, simplifica el diseño. Demuestre que son posibles simplificaciones similares para decremento unitario (suma de -1).

10.4 Sumadores como bloques constructores versátiles

Se puede usar un sumador binario de cuatro bits para realizar muchas funciones lógicas además de su función pretendida de suma. Por ejemplo, el sumador que implementa $(x_3x_2x_1x_0)_{\text{dos}} + (y_3y_2y_1y_0)_{\text{dos}} + c_0 = (c_4s_3s_2s_1s_0)_{\text{dos}}$ se puede usar como una XOR de cinco entradas para realizar la función $a \oplus b \oplus c \oplus d \oplus e$ al hacer $x_0 = a$,

$y_0 = b$, $c_0 = c$ (lo anterior conduce a $s_0 = a \oplus b \oplus c$), $x_1 = y_1 = d$ (esto último conduce a $c_2 = d$), $x_2 = e$, $y_2 = s_0$, y usar s_2 como la salida. Demuestre cómo se puede usar un sumador binario de cuatro bits como:

- Un circuito AND de cinco entradas
- Un circuito OR de cinco entradas
- Un circuito para realizar la función lógica de cuatro variables $ab \vee cd$
- Dos sumadores completos independientes de bit sencillo, cada uno con sus propias entradas y salidas de acarreo.
- Un circuito que multiplica por 15 para un número binario sin signo de dos bits $(u_1u_0)_{\text{dos}}$
- Un “contador paralelo” de cinco entradas que produce la suma de tres bits de cinco números de 1 bit

10.5 Números en complemento a dos

Pruebe lo siguiente para números x y y en complemento a 2 de k bits.

- Un corrimiento derecho aritmético de 1 bit de x siempre produce $\lfloor x/2 \rfloor$, sin importar el signo de x .
- En la suma $s = x + y$, ocurre desbordamiento ssi $c_{k-1} \neq c_k$.
- En un sumador que calcula $s = x + y$, pero no produce una señal de salida de acarreo c_k , el último se puede derivar externamente como $c_k = x_{k-1}y_{k-1} \vee s'_{k-1}$ ($x_{k-1} \vee y_{k-1}$).

10.6 Red de acarreo Brent-Kung

Dibuje un diagrama similar al de la figura 10.11 que corresponda a la red de acarreo de un sumador de 16 dígitos. *Sugerencia:* Use la construcción recursiva que se muestra en la figura 10.12.

10.7 Restador con anticipación de préstamo

Cualquier red de acarreo que produce los acarreos c_i con base en las señales g_i y p_i se puede usar, sin modi-

ficación, como un circuito de propagación de préstamo para encontrar los préstamos b_i .

- Defina las señales γ_i genera préstamo y π_i propaga préstamo para operandos de entrada binarios.
- Diseñe un circuito para calcular el dígito de diferencia d_i a partir de γ_i , π_i y el préstamo entrante b_i .

10.8 Incrementador con anticipación de acarreo

- En la sección 10.3 se observó que un incrementador, que calcula $x + 1$, es mucho más simple que un sumador y se presentó el diseño de un incrementador con acarreo en cascada. Diseñe un incrementador con anticipación de acarreo, saque ventaja de cualquier simplificación debida a que un operando sea 0.
- Repita la parte a) para un decrementador con anticipación de préstamo.

10.9 Árbitro de prioridad fija

Un árbitro de prioridad fija tiene k líneas de entrada de petición R_{k-1}, \dots, R_1, R_0 , y k líneas de salida garantizadas G_i . En cada ciclo de arbitraje, cuando mucho una de las señales garantizadas es 1 y ella corresponde a la señal de petición de prioridad más alta; es decir: $G_i = 1$ ssi $R_i = 1$ y $R_j = 0$ para $j < i$.

- Diseñe un árbitro síncrono con técnicas de acarreo en cascada. *Sugerencia:* Considere $c_0 = 1$ junto con reglas de propagación y aniquilación de acarreo; no hay generación de acarreo.
- Discuta el diseño de un árbitro más rápido con el uso de técnicas de anticipación de acarreo. Presente un diseño completo de árbitro de prioridad fija para $k = 64$.

10.10 Anticipación con bloques traslapantes

La ecuación $[i + 1, j] \notin [h, i] = [h, j]$, que se presentó en la sección 10.4, simboliza la función del operador *carry* que produce las señales g y p para el bloque $[h, j]$ compuesto de dos bloques más pequeños $[h, i]$ y $[i + 1, j]$. Demuestre que el operador *carry* todavía produce los valores correctos g y p para un bloque $[h, j]$ si se aplica a dos subbloques traslapantes $[h, i]$ y $[i - a, j]$, donde $a \geq 0$.

10.11 Red alterna con anticipación de acarreo

Una red de acarreo para $k = 2^a$ se puede definir de manera recursiva del modo siguiente. Una red de acarreo

de ancho 2 se construye con un solo operador *carry*. Para construir una red de acarreo de ancho 2^{h+1} a partir de dos redes de acarreo de ancho 2^h , se combina la última salida de la red inferior (parte menos significativa) con cada salida de la red superior usando un operador *carry*. Por ende, de acuerdo con dos redes de acarreo de ancho 2^h , una red de acarreo de ancho 2^{h+1} se puede sintetizar con el uso de 2^h bloques operadores *carry* adicionales.

- Dibuje una red de acarreo de ancho 16 con el uso de la construcción recursiva recién definida.
- Discuta cualesquier problemas que puedan surgir en la implementación del diseño de la parte a).

10.12 Sumador con salto de acarreo de bloque variable

Considere una red de acarreo de 16 bits construida mediante cuatro copias en cascada del bloque de salto de cuatro bits de la figura 10.7.

- Demuestre la trayectoria crítica de la red de acarreo resultante y derive su latencia en términos de retardos de compuerta.
- Demuestre que usar los anchos de bloque 3, 5, 5, 3, en lugar de 4, 4, 4, 4, conduce a una red de acarreo más rápida.
- Formule un principio general acerca de los sumadores con salto de acarreo de bloque variable con base en los resultados de la parte b).

10.13 Sumadores anchos contruidos a partir de estrechos

Suponga que tiene un suministro de sumadores de ocho bits que también proporcionan salidas para las señales de bloque g y p . Use una cantidad mínima de componentes adicionales para construir los tipos siguientes de sumadores para un ancho de $k = 24$ bits. Compare los sumadores de 24 bits resultantes en términos de latencia y costo, considere el costo y la latencia de una compuerta de a entradas como a unidades cada una y el costo y la latencia de un sumador de ocho bits como 50 y 10 unidades, respectivamente.

- Sumador con acarreo en cascada.
- Sumador con salto de acarreo con bloques de ocho bits.
- Sumador con selección de acarreo con bloques de ocho bits. *Sugerencia:* La señal *select* (selección)

para el multiplexor en los ocho bits superiores se puede derivar de c_8 y las dos salidas de acarreo en los ocho bits medios, si supone $c_8 = 0$ y $c_8 = 1$.

10.14 Sumadores para procesamiento de medios

Muchas señales de medios que procesan aplicaciones se caracterizan por operandos más estrechos (por decir, ocho o 16 bits de ancho). En tales aplicaciones, dos o cuatro elementos de datos se pueden empaquetar en una sola palabra de 32 bits. Sería más eficiente si los sumadores se diseñasen para manipular operaciones paralelas en muchos de tales operandos estrechos a la vez.

- Suponga que se quiere diseñar un sumador de 32 bits de modo que, cuando una señal especial *Halfadd* se postule, trate a cada uno de los dos operandos de entrada como un par de enteros independientes sin signo de 16 bits y sume las subpalabras correspondientes de los dos operandos, ello produce dos sumas sin signo de 16 bits. Para cada uno de los dos tipos de sumadores introducidos en este capítulo, muestre cómo se puede modificar el diseño para lograr la meta. Ignore el desbordamiento.
- ¿Cómo pueden modificarse los sumadores de la parte a) para acomodar cuatro sumas independientes de ocho bits, además de las dos sumas de 16 bits o una de 32 bits ya permitidas? *Sugerencia:* Considere usar una señal *quarteradd* que se postule cuando se deseen sumas independientes de ocho bits.
- En algunas aplicaciones es necesaria la *suma de saturación*, en la que la salida se fija al entero sin signo más grande cuando ocurre un desbordamiento. Discuta cómo se pueden modificar los diseños de los sumadores de las partes a) y b) para realizar suma de saturación siempre que se postule la señal de control *Saturadd*.

10.15 Detección de finalización de acarreo

Con el uso de dos redes de acarreo, una para la propagación de acarreos 1 y otra para la propagación de acarreos 0, se detecta la terminación del proceso de propagación de acarreo en un sumador con acarreo en cascada asíncrono. No tener que esperar el retardo para la propagación del acarreo de peor caso en cada suma resulta en un sumador con acarreo en cascada simple, con una latencia promedio que es competitiva con sumadores con anticipación de acarreo más complejos. En una po-

sición donde ambos bits operandos son 0 se “genera” un acarreo 0 y se propaga en la misma forma que un acarreo 1. El acarreo en la posición i se conoce cuando se postula la señal de acarreo 1 o 0. La propagación de acarreo está completa cuando se conocen los acarreos en todas las posiciones. Diseñe un sumador de detección de finalización de acarreo con base en la discusión previa.

10.16 Suma de tres operandos

- El cálculo de dirección en algunas máquinas puede requerir la suma de tres componentes: un valor base, un valor índice y un *offset*. Si supone valores base e índice sin signo de 32 bits, y un *offset* en complemento a dos de 16 bits, diseñe un circuito de cálculo de dirección rápido. *Sugerencia:* Vea la figura 9.3a.
- Si la suma de los tres componentes de dirección no se puede representar como un número sin signo de 32 bits, se postulará una señal de excepción de “dirección inválida”. Aumente el diseño de la parte a) para producir la señal de excepción requerida.

10.17 Desempacado de bits de un número

- Escriba un procedimiento MiniMIPS para tomar una palabra x de 32 bits como entrada y producir un arreglo de 32 palabras, que comience con una dirección específica en memoria, cuyos elementos sean 0 o 1 de acuerdo con el valor del correspondiente bit en x . El bit más significativo de x debe aparecer primero.
- Modifique el procedimiento de la parte a) de modo que también regrese el número de 1 en x . Esta operación, denominada *conteo de población*, es muy útil y a veces se proporciona como una instrucción de máquina.

10.18 Rotaciones y corrimientos

- Dibuje la contraparte de la figura 10.15 para rotaciones derecha e izquierda, en oposición a los corrimientos lógicos.
- Repita la parte a) para corrimientos aritméticos.

10.19 ALU para MiniMIPS-64

En el problema 8.11 se definió una versión de MiniMIPS compatible en retrospectiva de 64 bits. Discuta

cómo se debe modificar el diseño de ALU de la figura 10.19 para usar en esta nueva máquina de 64 bits.

10.20 Sumador decimal

Considere el diseño de un sumador decimal de 15 dígitos para números sin signo, donde cada dígito decimal está codificado en cuatro bits (para un ancho total de 60 bits).

- a) Diseñe los circuitos requeridos para las señales generar acarreo y propagar acarreo, si supone una codificación BCD para los operandos de entrada.

- b) Complete el diseño del sumador decimal de la parte a) para proponer un circuito de anticipación de acarreo y el circuito de cálculo de suma.

10.21 ALU multifunción

Para la ALU multifunción de la figura 10.19, especifique los valores de todas las señales de control para cada instrucción de tipo ALU en la tabla 6.2. Presente su respuesta en forma tabular, use “x” para entradas no importa.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Bren82] Brent, R. P. y H. T. Kung, “A Regular Layout for Parallel Adder”, *IEEE Trans. Computers*, vol. 31, núm. 3, pp. 260-264, marzo de 1982.
- [Goto02] Goto, G., “Fast Adders and Multipliers”, en *The Computer Engineering Handbook*, V. G. Oklobdzija, ed., pp. 9-22 a 9-41, CRC Press, 2002.
- [Lee02] Lee, R., “Media Signal Processing”, Section 39.1 en *The Computer Engineering Handbook*, V. G. Oklobdzija, ed., pp. 39-1 a 39-38, CRC Press, 2002.
- [Kore93] Koren, I., *Computer Arithmetic Algorithms*, Prentice Hall, 1993.
- [Parh00] Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.
- [Parh02] Parhami, B., “Number Representation and Computer Arithmetic”, en *Encyclopedia of Information Systems*, Academic Press, 2002, vol. 3, pp. 317–333.
- [Patt98] Patterson, D. A. y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2a. ed., 1998.
- [Swar90] Swartzlander, E. E., Jr, *Computer Arithmetic*, vols. I y II, IEEE Computer Society Press, 1990.

MULTIPLICADORES Y DIVISORES

“Así que esto es lo necesario para un hombre preocupado por la división y multiplicación con un entero... Al haber completado esto, ahora comenzaré a discutir la multiplicación de fracciones y su división, así como la extracción de raíces, si Dios lo quiere.”

Abu Jafar Muhammad al-Khwarizmi, Aritmética, ca. 830

“Al menos una buena razón para estudiar la multiplicación y la división es que existe un número infinito de formas de realizar estas operaciones; por tanto, existe un número infinito de doctorados (o visitas con gastos pagados a conferencias en Estados Unidos) a ganar a partir de la invención de nuevas formas de multiplicador.”

Alan Clements, Los principios del hardware computacional, 1986

TEMAS DEL CAPÍTULO

- 11.1** Multiplicación corrimiento-suma
- 11.2** Multiplicadores de hardware
- 11.3** Multiplicación programada
- 11.4** División corrimiento-resta
- 11.5** Divisores de hardware
- 11.6** División programada

La multiplicación y la división se usan en forma extensa, incluso en aplicaciones que no se asocian comúnmente con cálculos numéricos. Los principales ejemplos incluyen encriptación de datos para privacidad/seguridad, ciertos métodos de compresión de imágenes y representación gráfica. Los multiplicadores y divisores de hardware se han convertido en requisitos virtuales para todos, excepto los procesadores más limitados. Aquí se comienza con el algoritmo de multiplicación binaria corrimiento-suma, se muestra cómo se mapea el algoritmo a hardware y cómo se mejora aún más la rapidez de la unidad resultante. También se muestra cómo se puede programar, en una máquina que no tiene instrucción multiplicación, el algoritmo de la multiplicación corrimiento-suma. La discusión de la división sigue el mismo patrón: algoritmo binario corrimiento-resta, realización en hardware, métodos de aceleración y división programada.

■ 11.1 Multiplicación corrimiento-suma

Los multiplicadores de máquina más simples se diseñan para seguir una variante del algoritmo de la multiplicación con lápiz y papel que se muestra en la figura 11.1, donde cada hilera de puntos en la *matriz de bits de productos parciales* es todos 0 (si el correspondiente $y_i = 0$) o el mismo que x (si $y_i =$

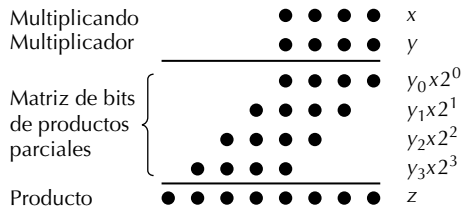


Figura 11.1 Multiplicación de números de cuatro bits en notación punto.

1). Cuando se realiza manualmente la multiplicación $k \times k$, se forman todos los k productos parciales y se suman los k números resultantes para obtener el producto p .

Para la ejecución en máquina es más fácil si un *producto parcial acumulado* se inicializa a $z^{(0)} = 0$, cada hilera de la matriz de bits se suma a aquél conforme se genera el término correspondiente y el resultado de la suma se corre hacia la derecha por un bit para lograr la alineación adecuada con el término siguiente, como se muestra en la figura 11.1. De hecho, esto último es exactamente como se realiza la *multiplicación programada* en una máquina que no tiene una unidad de multiplicación de hardware. La ecuación de recurrencia que describe el proceso es:

$$z^{(j+1)} = \underbrace{(z^{(j)} + y_j x 2^k)}_{\text{suma}} 2^{-1} \quad \text{con } z^{(0)} = 0 \quad y \quad z^{(k)} = z$$

|— corrimiento derecho —|

Puesto que, cuando se haya hecho, los corrimientos derechos habrán provocado que el primer producto parcial se multiplique por 2^{-k} , se premultiplica x por 2^k para desplazar el efecto de estos corrimientos derechos. Ésta no es una multiplicación real, sino que se hace para alinear x con la mitad superior del producto parcial acumulado de $2k$ bits en los pasos de suma. Este algoritmo de *multiplicación corrimiento-suma* se realiza directamente en hardware, como se verá en la sección 11.2. El corrimiento del producto parcial no necesita hacerse en un paso separado, sino que se puede incorporar en los alambres conectores que van de la salida del sumador al registro de doble ancho que retiene el producto parcial acumulado (figura 11.5).

Después de k iteraciones, la recurrencia de multiplicación conduce a:

$$z^{(k)} = 2^{-k} z^{(0)} + \sum_{j=0}^{k-1} (y_j x 2^j) = xy + 2^{-k} z^{(0)}$$

Por ende, si $z^{(0)}$ se inicializa a $2^k s$ (s rellenado con k ceros) en lugar de 0, se evaluará la expresión $xy + s$. Esta *operación multiplicar-sumar* es muy útil para muchas aplicaciones y se realiza sin costo adicional en comparación con la simple multiplicación corrimiento-suma.

La multiplicación en base r es similar a la anterior, excepto que todas las ocurrencias del número 2 en las ecuaciones se sustituyen con r .

Ejemplo 11.1: Multiplicación binaria y decimal

- Si supone operandos sin signo, realice la multiplicación binaria 1010×0011 , y muestre todos los pasos del algoritmo y productos parciales acumulados intermedios.
- Repita la parte a) para la multiplicación decimal 3528×4067 .

Solución: En lo que sigue, el término $2z^{(j+1)} = z^{(j)} + y_jx2^k$ se calcula con un dígito adicional en el extremo izquierdo para evitar perder la salida de acarreo (*carry-out*). Este dígito adicional se mueve hacia el producto parcial acumulado de ocho dígitos una vez que se corre a la derecha.

- a) Vea la figura 11.2a. El resultado se verifica al observar que representa $30 = 10 \times 3$.
- b) Vea la figura 11.2b. La única diferencia con la multiplicación binaria de la parte a) es que el término producto parcial y_jx10^4 necesita un dígito adicional. Por ende, si para este ejemplo se tuviese que dibujar el diagrama de notación punto de la figura 11.1, cada hilera de la matriz de bits de producto parcial contendría cinco puntos.

Posición	7	6	5	4	3	2	1	0
$x2^4$	1	0	1	0				
y					0	0	1	1
$z^{(0)}$	0	0	0	0				
$+y_0x2^4$	1	0	1	0				
$2z^{(1)}$	0	1	0	1	0			
$z^{(1)}$	0	1	0	1	0			
$+y_1x2^4$	1	0	1	0				
$2z^{(2)}$	0	1	1	1	1	0		
$z^{(2)}$	0	1	1	1	1	0		
$+y_2x2^4$	0	0	0	0				
$2z^{(3)}$	0	0	1	1	1	1	0	
$z^{(3)}$	0	0	1	1	1	1	0	
$+y_3x2^4$	0	0	0	0				
$2z^{(4)}$	0	0	0	1	1	1	1	0
$z^{(4)}$	0	0	0	1	1	1	1	0
a) Binario								

Posición	7	6	5	4	3	2	1	0
$x10^4$	3	5	2	8				
y					4	0	6	7
$z^{(0)}$	0	0	0	0				
$+y_0x10^4$	2	4	6	9	6			
$10z^{(1)}$	2	4	6	9	6			
$z^{(1)}$	2	4	6	9	6			
$+y_1x10^4$	2	1	1	6	8			
$10z^{(2)}$	2	3	6	3	7	6		
$z^{(2)}$	2	3	6	3	7	6		
$+y_2x10^4$	0	0	0	0	0			
$10z^{(3)}$	0	2	3	6	3	7	6	
$z^{(3)}$	0	2	3	6	3	7	6	
$+y_3x10^4$	1	4	1	1	2			
$10z^{(4)}$	1	4	3	4	8	3	7	6
$z^{(4)}$	1	4	3	4	8	3	7	6
b) Decimal								

Figura 11.2 Ejemplos de multiplicación paso a paso para números binario y decimal sin signo de cuatro dígitos.

Los números de magnitud con signo se pueden multiplicar con el uso del algoritmo corrimiento-suma ya discutido para multiplicar sus magnitudes y derivar por separado el signo del producto como XOR de los dos bits de signo entrantes.

Para entradas en complemento a 2, se requiere una simple modificación del algoritmo corrimiento-suma. Primero se nota que, si el multiplicador y es positivo, el algoritmo corrimiento-suma todavía funciona para el multiplicando negativo x . La razón se puede entender al examinar la figura 11.2a, donde $3x$ se calcula al evaluar $x + 2x$. Ahora, si x es negativo (en formato complemento a 2) se sumarían valores negativos x y $2x$, que deben conducir al resultado correcto $3x$. Todo lo que resta ahora es mostrar cómo se debe manipular un multiplicador y en complemento a 2 negativo.

Considere la multiplicación de x por $y = (1011)_{\text{compl } 2}$. Recuerde de la discusión en la sección 9.4 que el bit de signo se puede ver como ponderado negativamente. En consecuencia, el número y en complemento a 2 negativo es $-8 + 2 + 1 = -5$, mientras que, como número sin signo, es $8 + 2 + 1 = 11$. Así como se puede multiplicar x por $y = 11$ mediante el cálculo $8x + 2x + x$, se puede realizar la multiplicación por $y = -5$ al calcular $-8x + 2x + x$. Se ve que la única gran diferencia entre un multiplicador y sin signo y uno en complemento 2 es que, en el último paso de multiplicación, se debe restar, en lugar de sumar, el producto parcial $y_{k-1}x$.

Ejemplo 11.2: Multiplicación en complemento a 2

- a) Si supone operandos en complemento a 2, realice la multiplicación binaria 1010×0011 , y muestre todos los pasos del algoritmo y los productos parciales acumulados intermedios.
- b) Repita la parte a) para la multiplicación en complemento a 2 1010×1011 .

Solución: En lo que sigue, todos los productos parciales se escriben con un bit adicional a la izquierda para asegurar que la información de signo se preserve y manipula correctamente. En otras palabras, el multiplicando x de cuatro bits se debe escribir como $(11010)_{\text{compl } 2}$ o de otro modo su valor cambiará. Asimismo, la extensión de signo (corrimiento aritmético) se debe usar cuando $2z^{(i)}$ se corre a la derecha para obtener $z^{(i)}$.

- a) Vea la figura 11.3a. El resultado se verifica al observar que representa $-18 = (-6) \times 3$.
- b) Vea la figura 11.3b. El resultado se verifica al notar que representa $30 = (-6) \times (-5)$.

Posición	7	6	5	4	3	2	1	0
$x2^4$	1	0	1	0				
y					0	0	1	1
$z^{(0)}$	0	0	0	0	0			
$+y_0x2^4$	1	1	0	1	0			
$2z^{(1)}$	1	1	0	1	0			
$z^{(1)}$	1	1	1	0	1	0		
$+y_1x2^4$	1	1	0	1	0			
$2z^{(2)}$	1	0	1	1	1	0		
$z^{(2)}$	1	1	0	1	1	1	0	
$+y_2x2^4$	0	0	0	0	0			
$2z^{(3)}$	1	1	0	1	1	1	0	
$z^{(3)}$	1	1	1	0	1	1	1	0
$+(-y_3x2^4)$	0	0	0	0	0			
$2z^{(4)}$	1	1	1	0	1	1	1	0
$z^{(4)}$	1	1	1	0	1	1	1	0

a) Multiplicador y positivo

Posición	7	6	5	4	3	2	1	0
$x2^4$	1	0	1	0				
y					1	0	1	1
$z^{(0)}$	0	0	0	0	0			
$+y_0x2^4$	1	1	0	1	0			
$2z^{(1)}$	1	1	0	1	0			
$z^{(1)}$	1	1	1	0	1	0		
$+y_1x2^4$	1	1	0	1	0			
$2z^{(2)}$	1	0	1	1	1	0		
$z^{(2)}$	1	1	0	1	1	1	0	
$+y_2x2^4$	0	0	0	0	0			
$2z^{(3)}$	1	1	0	1	1	1	0	
$z^{(3)}$	1	1	1	0	1	1	1	0
$+(-y_3x2^4)$	0	0	1	1	0			
$2z^{(4)}$	0	0	0	1	1	1	1	0
$z^{(4)}$	0	0	0	1	1	1	1	0

b) Multiplicador y negativo

Figura 11.3 Ejemplos de multiplicación paso a paso para números en complemento a 2 de cuatro dígitos.

11.2 Multiplicadores de hardware

El algoritmo corrimiento-suma de la sección 11.1 se puede convertir directamente en el multiplicador de hardware que se muestra en la figura 11.4. Existen k ciclos para multiplicación $k \times k$. En el j -ésimo ciclo, x o 0 se suman a la mitad superior del producto parcial de doble ancho, dependiendo del j -ésimo bit y_j del multiplicador, y el nuevo producto parcial se corre a la derecha antes del comienzo del ciclo siguiente. El multiplexor de la figura 11.4 permite que la resta se realice en el último ciclo, como lo requiere la multiplicación en complemento a 2. En un multiplicador que se use sólo con operandos sin signo, el multiplexor se puede sustituir por un arreglo de compuertas AND que en conjunto multipliquen el bit y_j por el multiplicando x .

En lugar de almacenar y en un registro separado, como se muestra en la figura 11.4, se le puede retener en la mitad inferior del registro de producto parcial. Lo anterior se justifica porque como el producto parcial se expande para ocupar más bits de la mitad inferior en el registro, los bits del multiplicador se retiran debido a que se corren a la derecha. Los ejemplos de multiplicación binaria de las figuras 11.2a y 11.3 muestran cómo la mitad inferior del registro de producto parcial inicialmente no se usa y luego los bits se corren hacia él a la tasa de uno por ciclo.

En vez de tratar el corrimiento como una tarea que se debe realizar después de que la salida del sumador se haya almacenado en el registro de producto parcial, se le puede incorporar en el aspecto de cómo los datos se almacenan en el registro. Recuerde de la figura 2.1 que para almacenar una palabra en un registro, los bits de ésta se deben suministrar a los *flip-flops* individuales que comprenden el registro. Es fácil mover los bits hacia la derecha una posición antes de suministrarlos al registro y valorar la señal *load* (carga). La figura 11.5 muestra cómo se hacen las conexiones a las líneas de datos de entrada al registro para lograr el objetivo de carga con corrimiento a la derecha.

La multiplicación en base 2 se puede implementar de manera muy directa e involucra poca complejidad de hardware adicional en una ALU que ya tenga sumador/restador. Sin embargo, los k ciclos de reloj necesarios para realizar una multiplicación $k \times k$ hace que esta operación sea mucho más lenta que las operaciones suma y lógico/corrimiento, que toman apenas un solo ciclo de reloj. Una forma de acelerar el hardware de multiplicación consiste en realizar la multiplicación en base 4 o bases más altas.

En base 4, cada ciclo de operación se hace cargo de dos bits del multiplicador y corta el número de ciclos a la mitad. Al considerar, por simplicidad, la multiplicación sin signo los dos bits del multiplica-

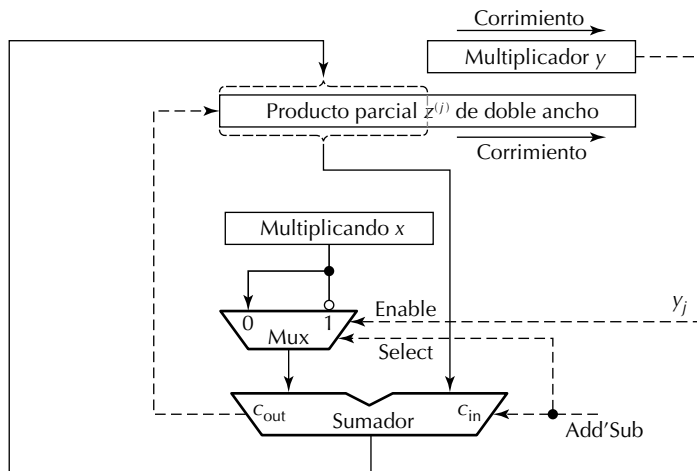


Figura 11.4 Multiplicador de hardware con base en el algoritmo corrimiento-suma.

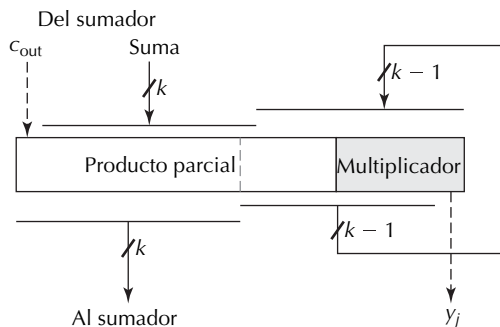


Figura 11.5 Corrimiento incorporado en las conexiones hacia el registro de producto parcial y no como fase separada.

El producto parcial acumulado en un ciclo específico se usan para seleccionar uno de cuatro valores a sumar al producto parcial acumulado: 0 , x , $2x$ (x corrida), $3x$ (precalculado mediante la suma $2x + x$ y almacenada en un registro antes de que comiencen las iteraciones). En implementaciones reales, el precálculo de $3x$ se evita mediante métodos de recodificación especiales, como la recodificación Booth. Adicionalmente, el producto parcial acumulado se mantiene en forma de almacenamiento de acarreo, lo que hace muy rápida la suma del siguiente producto parcial a él (figura 9.3a). De esta forma sólo se requiere una suma regular en el último paso, ello conduce a una aceleración significativa. Los detalles de la implementación para *multiplicadores de base alta* y recodificados no son propósito de estudio en este texto, pero se les encuentra en libros de aritmética computacional [Parh00].

Las unidades de multiplicación de hardware rápidas en procesadores de alto rendimiento se basan en diseños de *árboles multiplicadores*. En lugar de formar los productos parciales uno a la vez en base 2 o h a la vez en base 2^h , se pueden formar todos ellos simultáneamente, y se reduce el problema de multiplicación a una suma de n operandos, donde $n = k$ en base 2, $n = k/2$ en base 4, etc. Por ejemplo, la multiplicación 32×32 se convierte en un problema de suma de 32 operandos en base 2 o un problema de suma de 16 operandos en base 4. En los árboles multiplicadores, los n operandos así formados se suman en dos etapas. En la etapa 1, un árbol construido con *sumadores con almacenamiento de acarreo* se usa para reducir los n operandos a dos operandos que tienen la misma suma que los n números originales. Un sumador con almacenamiento de acarreo (figura 9.3a) reduce tres valores a dos valores, por un factor de reducción de 1.5, ello lleva a un circuito de $\lceil \log_{1.5}(n/2) \rceil$ niveles para reducir n números a 2. Los dos números derivados así se suman mediante un sumador rápido de tiempo logarítmico, lo cual conduce a una latencia logarítmica global para el multiplicador (figura 11.6a).

El *multiplicador de árbol completo* de la figura 11.6a es complejo y su rapidez puede no ser necesaria para todas las aplicaciones. En tales casos se pueden implementar *multiplicadores de árbol parcial* más económicos. Por ejemplo, si casi la mitad de los productos parciales se acomodan por la parte de árbol, entonces se pueden usar dos pases a través del árbol para formar los dos números que representan el producto deseado, y los resultados de la primera pasada se retroalimentan a las entradas y se combinan con el segundo conjunto de productos parciales (figura 11.6b). Un multiplicador de árbol parcial se puede ver como un multiplicador de base (muy) alta. Por ejemplo, si 12 productos parciales se combinan en cada pasada, entonces se realiza efectivamente una multiplicación de base 2^{12} .

Un *arreglo multiplicador* usa el mismo esquema de computación en dos etapas de un árbol multiplicador, con la diferencia de que el árbol de sumadores con almacenamiento de acarreo es unilateral (tiene la máxima profundidad posible de k para la multiplicación $k \times k$) y el sumador final es del tipo acarreo en cascada (muy lento). En la figura 11.7 se muestra un arreglo multiplicador 4×4 , donde las celdas HA y FA son medio sumadores y sumadores completos definidos en las figuras 10.1 y 10.2, respectivamente; las celdas MA son sumadores completos modificados, una de cuyas entradas se forma internamente como la AND lógica de x_i y y_j .

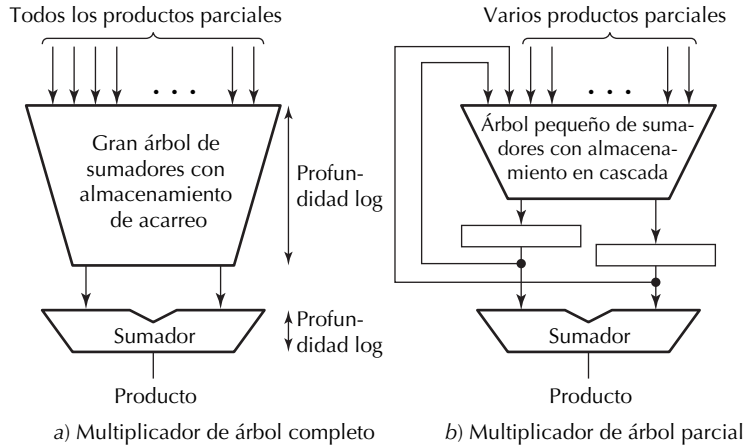


Figura 11.6 Diagramas esquemáticos de multiplicadores de árbol completo y árbol parcial.

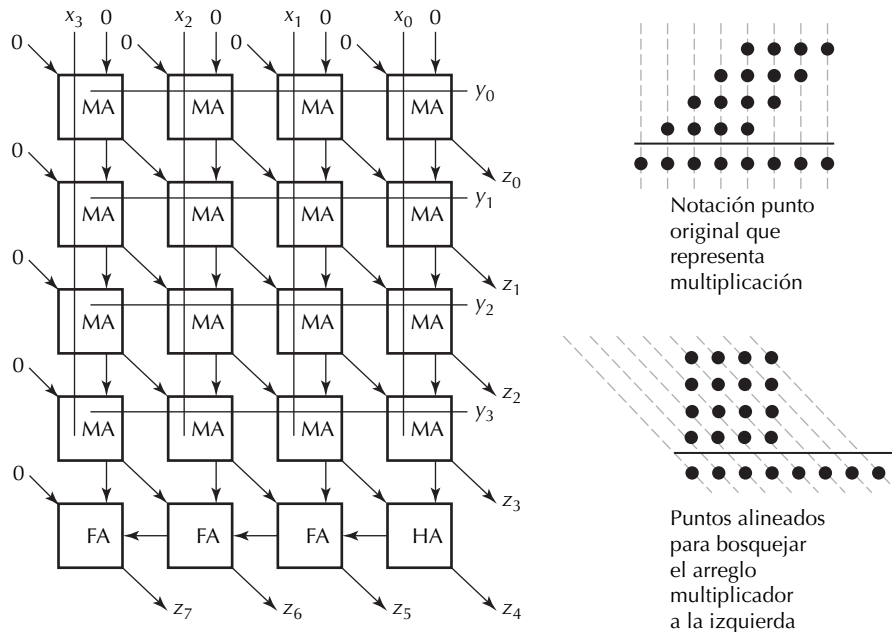


Figura 11.7 Arreglo multiplicador de operandos sin signo de cuatro bits.

Probablemente se pregunta por qué tal tipo de árbol multiplicador lento es de algún interés. La respuesta es que los arreglos multiplicadores son bastante adecuados para la realización VLSI, dado su diseño enormemente regular y su eficiente patrón de alambrado. También se pueden encauzar fácilmente con la inserción de *latches* entre algunas de las hileras de celdas; por tanto, permite que muchas multiplicaciones se puedan realizar en la misma estructura de hardware.

11.3 Multiplicación programada

MiniMIPS tiene dos instrucciones *multiply* que realizan multiplicaciones en complemento a 2 y sin signo, respectivamente. Ellas son:

```
mult    $s0, $s1    # fija Hi,Lo en ($s0) × ($s1); con signo
multu   $s2, $s3    # fija Hi,Lo en ($s2) × ($s3); sin signo
```

Dichas instrucciones dejan el producto de doble ancho en los registros especiales Hi y Lo, donde Hi retiene la mitad superior y Lo la mitad inferior del producto de 64 bits. Por qué, al igual que otras instrucciones de tipo R de MiniMIPS, el producto no se coloca en un par de registros en el archivo registro, se volverá claro conforme se discuta la ejecución de instrucciones y la secuenciación de control en la parte cuatro de este libro. Por ahora sólo se menciona que la razón se relaciona con la latencia mucho más grande de la multiplicación en relación con las operaciones suma y lógica/corrimiento. Puesto que el producto de dos números de 32 bits siempre es representable en 64 bits, no puede haber desbordamiento en algún tipo de multiplicación.

Para permitir el acceso a los contenidos de Hi y Lo, se proporcionan las siguientes dos instrucciones para copiar sus contenidos en los registros deseados en el archivo registro general:

```
mfhi    $t0          # fija $t0 en (Hi)
mflo    $t1          # fija $t1 en (Lo)
```

Para la representación en máquina de estas cuatro instrucciones, vea la sección 6.6.

Observe que, si se desea un producto de 32 bits, se puede recuperar del registro Lo. Sin embargo, se debe examinar el contenido de Hi para estar seguro de que el producto correcto es representable en 32 bits; esto es, no superó el rango de números de 32 bits.

Ejemplo 11.3: Uso de multiplicación en programas MiniMIPS Muestre cómo obtener el producto de 32 bits de enteros con signo de 32 bits que se almacenan en los registros \$s3 y \$s7, y coloque el producto en el registro \$t3. Si el producto no es representativo en 32 bits, el control se debe transferir a movfl. Suponga que los registros \$t1 y \$t2 están disponibles para resultados anulados si fuera necesario.

Solución: Si el producto de enteros con signo debe encajar en una palabra de 32 bits, se espera que el registro especial Hi retenga 32 bits idénticos iguales al bit de signo de Lo (si fuese una multiplicación sin signo, Hi debería verificarse para 0). Por tanto, si el valor en Lo es positivo, Hi debe retener 0 y si el valor en Lo es negativo, Hi debe retener -1, que tiene la representación todos 1.

```
mult    $s3,$s7      # producto formado en Hi,Lo
mfhi    $t2          # copia mitad superior del producto en $t2
mflo    $t3          # copia mitad inferior del producto en $t3
slt     $t1,$t3,$zero # fija (LSB de) $t1 al signo de Lo
sll     $t1,$t1,31    # fija bit de signo de $t1 al signo de Lo
sra     $t1,$t1,31    # fija $t1 a todos 0/1 vía corrimiento arit
bne     $t1,$t2,movfl # si (Hi) ≠ ($t1), se tiene desbordamiento
```

En las máquinas que no tienen una instrucción *multiply* soportada por hardware, la multiplicación se puede realizar en software mediante el algoritmo corrimiento-suma de la sección 11.1. Es conveniente desarrollar tal programa para MiniMIPS porque ayuda a comprender mejor el algoritmo. En lo que sigue, se considera la multiplicación sin signo y se deja al lector desarrollar la versión con signo.

Ejemplo 11.4: Multiplicación corrimiento-suma de números sin signo Con el algoritmo corrimiento-suma, defina un procedimiento MiniMIPS que multiplique dos números sin signo (pasados a él en los registros \$a0 y \$a1) y deje el producto de doble ancho en \$v0 (parte alta) y \$v1 (parte baja).

Solución: El siguiente procedimiento, llamado *shamu* por “multiplicación corrimiento-suma” usa tanto instrucciones como pseudoinstrucciones. Los registros Hi y Lo, que retienen las mitades superior e inferior del producto parcial acumulado *z*, se representan mediante \$v0 y \$v1, respectivamente. El multiplicando *x* está en \$a0 y el multiplicador *y* está en \$a1. El registro \$t2 se usa como un contador que se inicializa a 32 y decrementa por 1 en cada iteración hasta que alcanza 0. El *j*-ésimo bit de *y* se aísla en \$t1 mediante corrimiento derecho repetido y se busca su LSB después de cada corrimiento. El registro \$t1 también se usa para aislar el LSB de Hi de modo que se pueda correr en Lo durante corrimiento derecho. Finalmente, \$t0 se usa para calcular la salida de acarreo de la suma que se debe correr en Hi durante corrimientos derechos. El uso de registro en este ejemplo se muestra en la figura 11.8 para fácil referencia.

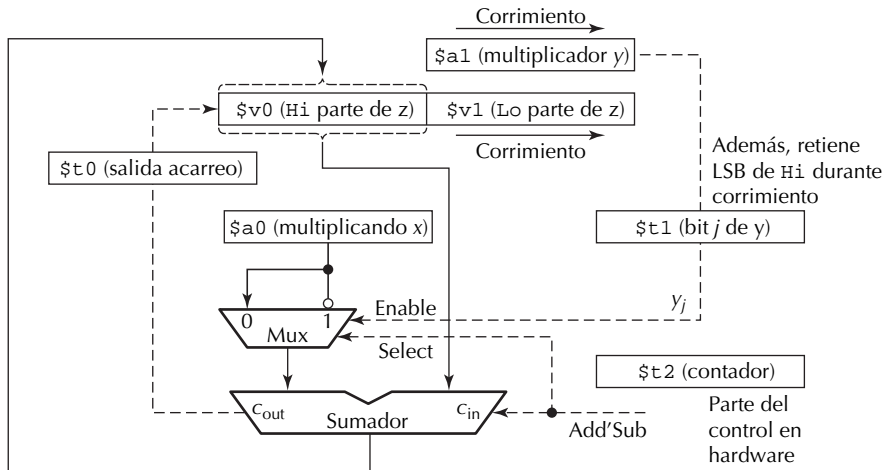


Figura 11.8 Uso de registro para multiplicación programada sobreimpresa en el diagrama de bloques para un multiplicador de hardware.

```
shamu: move $v0,$zero      # inicializa Hi a 0
       move $v1,$zero      # inicializa Lo a 0
       addi $t2,$zero,32   # inicializa contador de repetición a 32
mloop: move $t0,$zero      # fija salida acarreo a 0 en caso de no
                           suma
```

```

        move $t1,$a1      # copia ($a1) en $t1
        srl  $a1,1        # divide a la mitad el valor sin signo en
                           # $a1
        subu $t1,$t1,$a1  # resta ($a1) de ($t1) dos veces para
        subu $t1,$t1,$a1  # obtener LSB de ($a1), o y[j], en $t1
        beqz $t1,noadd    # no necesita suma si y[j]=0
        addu $v0,$v0,$a0  # suma x a parte superior de z
        sltu $t0,$v0,$a0  # forma salida acarreo de suma en $t0
noadd:  move $t1,$v0      # copia ($v0) en $t1
        srl  $v0,1        # divide a la mitad el valor sin signo en
                           # $v0
        subu $t1,$t1,$v0  # resta ($v0) de ($t1) dos veces para
        subu $t1,$t1,$v0  # obtener LSB de Hi en $t1
        sll  $t0,$t0,31   # salida acarreo convertida a 1 en MSB de
                           # $t0
        addu $v0,$v0,$t0  # corrido derecha $v0 corregido
        srl  $v1,1        # divide a la mitad el valor sin signo en
                           # $v1
        sll  $t1,$t1,31   # LSB de Hi convertido a 1 en MSB de $t1
        addu $v1,$v1,$t1  # corrido derecha $v1 corregido
        addi $t2,$t2,-1   # decremento de contador repetición en 1
        bne  $t2,$zero,mloop # si contador > 0, repetir ciclo
                           # multiplicación
        jr   $ra          # regresa al programa llamante

```

Observe que, dado que la salida de acarreo de la suma no se registra en MiniMIPS, se vislumbró un esquema para derivarla en `$t0` al notar que la salida de acarreo es 1 (un desbordamiento de suma sin signo) ssi la suma es menor que cualquier operando.

Cuando un multiplicando x se debe multiplicar por un multiplicador constante a , usar la instrucción *multiply*, o su versión en software del ejemplo 11.4, puede no ser la mejor opción. La multiplicación por una potencia de 2 se puede efectuar mediante corrimiento, que es más rápido y más efectivo que una instrucción *multiply* de quemado completo. En este contexto, la multiplicación por otras constantes pequeñas también se puede realizar mediante las instrucciones *shift* y *add*. Por ejemplo, para calcular $5x$, se puede formar $4x$ por una instrucción *left-shift* (corrimiento izquierdo) y luego sumar x al resultado. En muchas máquinas, una instrucción *shift* y otra *add* toman menos tiempo que una instrucción *multiply*. Como otro ejemplo, $7x$ se puede calcular como $8x - x$, aunque en este caso existe el peligro de encontrar desbordamiento al calcular $8x$ aun cuando $7x$ por sí mismo esté dentro del rango de representación de la máquina. Los compiladores más modernos evitan devolver las instrucciones *multiply* cuando una secuencia corta de instrucciones *add/subtract/shift* pueden lograr la misma meta.

■ 11.4 División corrimiento-resta

Al igual que los multiplicadores, las máquinas divisoras más simples se diseñan para seguir una variante del algoritmo de lápiz y papel que se muestra en la figura 11.9. Cada hilera de puntos en la matriz

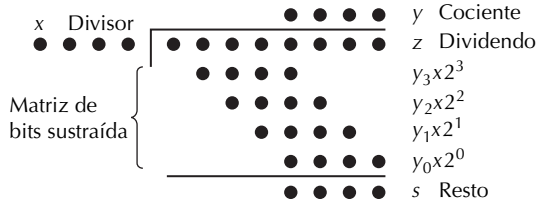


Figura 11.9 División de un número de ocho bits por un número de cuatro bits en notación punto.

de bits sustraída de la figura 11.9 es o todos 0 (si la correspondiente $y_i = 0$) o igual que x (si $y_i = 1$). Cuando se realiza manualmente una división $2k/k$, los términos sustraídos se forman uno a la vez mediante la “adivinación” del valor del siguiente *dígito cociente*, restar el término adecuado (0 o una versión con corrimiento adecuado para x) del *resto parcial*, que se inicializa al valor del dividendo z , y se procede hasta que todos los k bits del cociente y se hayan determinado. En este momento, el resto parcial se convierte en el resto final s .

Para implementación en hardware o software, se usa una ecuación de recurrencia que describe el proceso anterior:

$$z^{(j)} = 2z^{(j-1)} - y_{k-j}x2^k \quad \text{con } z^{(0)} = z \text{ y } z^{(k)} = 2^k s$$

|— corrimiento —|
|— sustraer —|

Puesto que, para el momento en que se realice, los corrimientos izquierdos habrán provocado que el resto parcial se multiplique por 2^k , el resto verdadero se obtiene al multiplicar el resto parcial final por 2^{-k} (corriéndolo a la derecha por k bits). Esto se justifica como sigue:

$$z^{(k)} = 2^k z - \sum_{j=1}^k 2^{k-j} (y_{k-j} x 2^k) = 2^k (z - xy) = 2^k s$$

Al igual que en el caso de productos parciales para la multiplicación, el corrimiento del resto parcial no se necesita realizar en un paso separado, sino que se puede incorporar en los alambres conectores que van de la salida del sumador al registro de doble ancho que retiene el resto parcial (sección 11.5).

La división en base r es similar a la división binaria, excepto que todas las ocurrencias del número 2 en las ecuaciones se sustituyen por r .

Ejemplo 11.5: División entera y fraccional

- Si supone operandos sin signo, realice la división binaria 0111 0101/1010, y muestre todos los pasos del algoritmo y los restos parciales.
- Repita la parte a) para la división decimal fraccional .1435 1502/.4067.

Solución: En lo que sigue, el término $2z^{(j)}$ se forma con un dígito adicional en el extremo izquierdo para evitar que se pierda el dígito adicional creado al duplicar.

- Vea la figura 11.10a. Los resultados se verifican al observar que representan $117 = 10 \times 11 + 7$.
- Vea la figura 11.10b. Las únicas diferencias de la división binaria de la parte a) son que el dividendo x no se premultiplica por una potencia de la base y los términos sustraídos $y_j x$ necesitan un dígito adicional. Por ende, si se quisiera dibujar el diagrama de notación punto de la figura 11.9 para este ejemplo, cada hilera de la matriz de bits sustraída contendría cinco puntos.

Posición	7	6	5	4	3	2	1	0		Posición	-1	-2	-3	-4	-5	-6	-7	-8	
z	0	1	1	1	0	1	0	1		z	.	1	4	3	5	1	5	0	2
$x2^4$	1	0	1	0						x	.	4	0	6	7				
$z^{(0)}$	0	1	1	1	0	1	0	1		$z^{(0)}$.	1	4	3	5	1	5	0	2
$2z^{(0)}$	0	1	1	1	0	1	0	1		$10z^{(0)}$	1	.	4	3	5	1	5	0	2
$-y_3x2^4$	1	0	1	0					$y_3 = 0$	$-y_{-1}x$	1	.	2	2	0	1			$y_{-1} = 3$
$z^{(1)}$	0	1	0	0	1	0	1			$z^{(1)}$.	2	1	5	0	5	0	2	
$2z^{(1)}$	0	1	0	0	1	0	1			$10z^{(1)}$	2	.	1	5	0	5	0	2	
$-y_2x2^4$	0	0	0	0					$y_2 = 0$	$-y_{-2}x$	2	.	0	3	3	5			$y_{-2} = 5$
$z^{(2)}$	1	0	0	1	0	1				$z^{(2)}$.	1	1	7	0	0	2		
$2z^{(2)}$	1	0	0	1	0	1				$10z^{(2)}$	1	.	1	7	0	0	2		
$-y_1x2^4$	1	0	1	0					$y_1 = 1$	$-y_{-3}x$	0	.	8	1	3	4			$y_{-3} = 2$
$z^{(3)}$	1	0	0	0	1					$z^{(3)}$.	3	5	6	6	2			
$2z^{(3)}$	1	0	0	0	1					$10z^{(3)}$	3	.	5	6	6	2			
$-y_0x2^4$	1	0	1	0					$y_0 = 0$	$-y_{-4}x$	3	.	2	5	3	6			$y_{-4} = 8$
$z^{(4)}$	0	1	1	1						$z^{(4)}$.	3	1	2	6				
s					0	1	1	1		s	.	0	0	0	0	3	1	2	6
y					1	0	1	1		y	.	3	5	2	8				

a) Binario entero

b) Decimal fraccional

Figura 11.10 Ejemplos paso a paso de división para números enteros binarios sin signo y números fraccionales decimales de 8/4 dígitos.

Ejemplo 11.6: División con operandos del mismo ancho Con frecuencia, la división z/x se realiza con operandos del mismo ancho en vez de con z siendo el doble de ancho que x . Sin embargo, el algoritmo es el mismo.

- a) Si supone operandos sin signo, realice la división binaria 1101/0101, muestre todos los pasos del algoritmo y restos parciales.
- b) Repita la parte a) para la división binaria fraccional .0101/.1101.

Solución

- a) Vea la figura 11.11a. Observe que, dado que z tiene cuatro bits de ancho, el corrimiento izquierdo nunca puede producir un bit distinto de cero en la posición 8; así que, a diferencia del ejemplo en la figura 11.10, no es necesario un bit adicional en la posición 8. Los resultados se verifican tomando en cuenta que ellos representan $13 = 5 \times 2 + 3$.
- b) Vea la figura 11.11b. Las únicas diferencias de la división binaria de la parte a) son que el dividendo x no se premultiplica por una potencia de la base y que tanto los restos parciales con corrimiento como los términos $y_{-j}x$ sustraídos necesitan un bit adicional a la izquierda (posición 0). Los resultados se verifican al observar que ellos representan $5/16 = (13/16) \times (6/16) + 2/256$.

De los ejemplos anteriores se ve que las magnitudes del cociente y el resto s no se alteran por los signos de entrada y que los signos y y s se derivan fácilmente de z y x . Por tanto, una forma de hacer división con signo es a través de un algoritmo indirecto que convierte los operandos en valores sin signo y , al final, explicar los signos mediante el ajuste de los bits de signo o vía complementación. Éste es el método de elección con el algoritmo de división restauradora, discutida en esta sección. La división directa de números con signo es posible mediante el algoritmo de división no restauradora, cuya discusión no se trata en este libro. Los lectores interesados se pueden remitir a cualquier texto acerca de aritmética computacional [Parh00].

11.5 Divisores de hardware

El algoritmo corrimiento-resta de la sección 11.4 se puede convertir en el divisor de hardware que se muestra en la figura 11.12. Existen k ciclos para división $2k/k$ o k/k . En el j -ésimo ciclo, x se resta de la mitad superior del resto parcial de doble ancho. El resultado, conocido como la *diferencia de prueba*, se carga en el registro de resto parcial sólo si es positivo. El signo de la diferencia de prueba dice si 1 es la elección correcta para el siguiente dígito cociente y_{k-j} o es demasiado grande (0 es la elección correcta). Observe que la diferencia de prueba es positiva si el MSB del resto parcial previo es 1, ello propicia que el resto parcial corrido sea lo suficientemente grande para garantizar una diferencia positiva, o de otro modo si la c_{out} del sumador es 1. Ésta es la razón para proporcionar estos dos bits al bloque selector de dígito cociente.

El multiplexor de la figura 11.12 permite que se realice o suma o resta en cada ciclo. La suma de x nunca se requiere para el *algoritmo de división restauradora* presentada en la sección 11.4. El adjetivo “restauradora” significa que siempre que la diferencia de prueba se vuelva negativa, se toma como indicador de que el siguiente dígito cociente es 0 y la diferencia de prueba no se carga en el registro resto parcial, ello provoca que el valor original permanezca intacto al final del ciclo. En la *división no restauradora*, que no se cubre en este libro, la diferencia calculada, positiva o negativa, se almacena como el resto parcial, que, por tanto, no se restaura a su valor correcto cuando se vuelve negativo. Sin

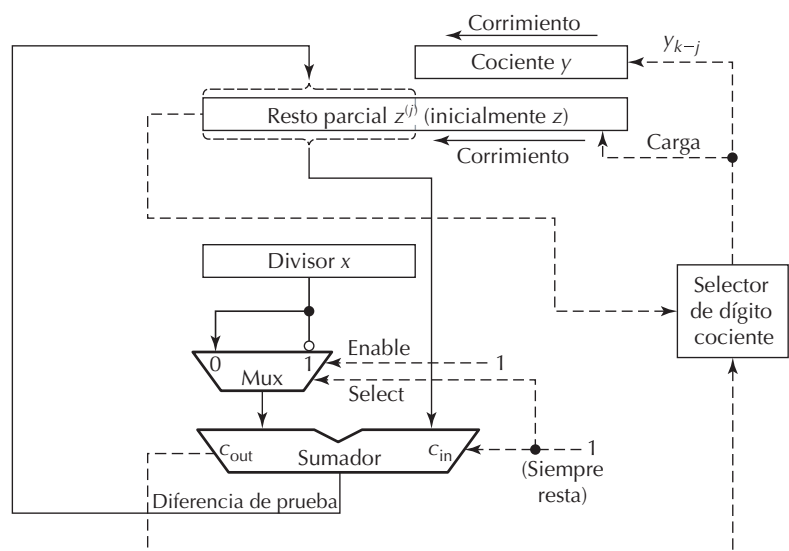


Figura 11.12 Divisor de hardware con base en el algoritmo corrimiento-resta.

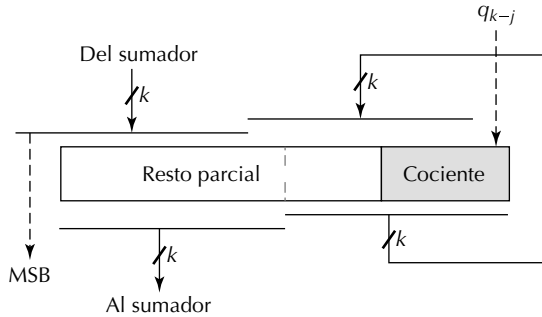


Figura 11.13 Corrimiento incorporado en las conexiones al registro resto parcial en lugar de como fase separada.

embargo, acciones apropiadas en pasos subsecuentes conducirán al resultado correcto. Estas acciones correctivas requieren que x se sume al, en lugar de restar del, resto parcial. El enfoque no restaurador hace el circuito de control más simple y tiene el beneficio colateral de interpretar una posible división con signo directa.

En lugar de insertar los bits de y en un registro separado, como se muestra en la figura 11.12, se les insertará en la mitad inferior del registro de resto parcial. Lo anterior sucede porque conforme el resto parcial se corre a la izquierda, las posiciones de bit en el extremo derecho del registro se liberan y se pueden usar para almacenar los bits cociente según se desarrollan. Los ejemplos de división en la figura 11.10 claramente muestran cómo la mitad inferior del registro de resto parcial, que está totalmente ocupado en el principio, se libera a la tasa de un bit/dígito por ciclo.

En lugar de tratar el corrimiento como una tarea que se debe realizar después de que la salida del sumador se almacene en el registro de resto parcial, se puede incorporar el corrimiento en cómo los datos se almacenan en el registro. Recuerde de la figura 2.1 que, para almacenar una palabra en un registro, los bits de la palabra se deben suministrar al conjunto de *flip-flops* que comprenden el registro. Es fácil mover los bits a la izquierda por una posición antes de suministrarlos al registro y valorar la señal *load* (carga). La figura 11.13 muestra cómo se hacen las conexiones a las líneas de datos de entrada del registro para lograr el objetivo de cargar corrido izquierdo.

Una comparación de las figuras 11.4 y 11.12 revela que los multiplicadores y divisores son bastante similares y se pueden implementar con hardware compartido dentro de una ALU que ejecuta diferentes operaciones con base en un código de función proporcionado externamente. De hecho, en virtud de que la *raíz cuadrada* es muy similar a la división, es común implementar una sola unidad que realice las operaciones de multiplicación, división y raíz cuadrada.

Al igual que en el caso de los multiplicadores, los *divisores de base alta* aceleran el proceso de división mediante la producción de muchos bits del cociente en cada ciclo. Por ejemplo, en base 4 cada ciclo de operación genera dos bits del cociente; por tanto, corta el número de ciclos a la mitad. Sin embargo, hay una complicación que hace al proceso un poco más difícil que la multiplicación en base 4: puesto que los dígitos en base 4 tienen cuatro posibles valores, no se puede usar un esquema de selección igual que el de base 2 (es decir, intentar 1 y elegir 0 si no funciona) para la selección de dígito cociente en base 4. Los detalles de cómo se resuelve este problema y cómo, para productos parciales en la multiplicación, el resto parcial en la división se puede retener en la forma de almacenamiento de acarreo, no se tratan en este libro.

Mientras que no hay contraparte a los multiplicadores de árbol para realizar la división, existen *arreglos divisores* y estructuralmente son similares a los arreglos multiplicadores. La figura 11.14 muestra un arreglo divisor que divide un dividendo z de ocho bits por un divisor x de cuatro bits, ello produce un cociente y de cuatro bits y un resto s de cuatro bits. Las celdas MS de la figura 11.14 son

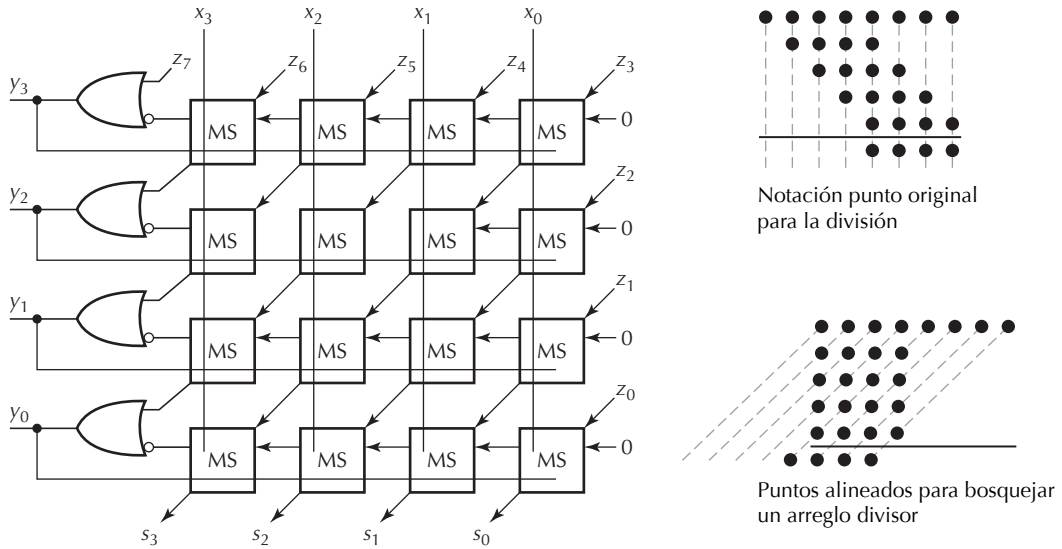


Figura 11.14 Arreglo divisor para enteros sin signo de 8/4 bits.

restadores completos modificados. Ellos reciben un bit de entrada de préstamo de la derecha y dos bits operandos de arriba, restan el operando vertical del operando diagonal y producen un bit diferencia que baja diagonalmente y un bit de salida de préstamo que va hacia la izquierda. Existe un multiplexor en la salida diagonal que permite pasar la diferencia si y_i , presentada a las celdas como una línea de control horizontal proveniente de la izquierda, es 1 y deja pasar z si $y_i = 0$. Esto último sirve a la función de usar la diferencia de prueba como el siguiente resto parcial o restaurar el resto parcial al valor anterior.

También es posible realizar la división con el uso de una secuencia de multiplicaciones en lugar de sumas. Aun cuando las multiplicaciones sean más lentas y más complejas que las sumas, se puede ganar una ventaja sobre los esquemas aditivos, pues se necesitan pocas multiplicaciones para realizar la división. De este modo, en tales *algoritmos de división de convergencia* se negocia la complejidad de las iteraciones por menos iteraciones. En este sentido, en la división de convergencia se deriva el cociente $y = z/x$ al calcular primero $1/x$ y luego multiplicarlo por z . Para calcular $1/x$, se comienza con una aproximación burda obtenida sobre la base de unos cuantos bits de orden superior de x , usando un circuito lógico especialmente diseñado o una pequeña tabla de referencia. La aproximación $u^{(0)}$ tendrá un error relativo pequeño de, por decir, ϵ , ello significa que $u^{(0)} = (1 + \epsilon)(1/x)$. Entonces se obtienen aproximaciones sucesivamente mejores a $1/x$ con el uso de una aproximación $u^{(i)}$ en la fórmula

$$u^{(i+1)} = u^{(i)} \times (2 - u^{(i)} \times x)$$

De este modo, cada iteración para refinar el valor de $1/x$ involucra dos multiplicaciones y una resta. Si la aproximación inicial es exacta a, por decir, ocho bits, la siguiente será exacta a 16 bits y la que sigue a 32 bits. Por tanto, entre dos y cuatro iteraciones son suficientes en la práctica. Los detalles para implementar tales esquemas de división de convergencia y el análisis del número de iteraciones necesarias se pueden encontrar en libros acerca de aritmética computacional [Parh00].

11.6 División programada

MiniMIPS tiene dos instrucciones del tipo `divide` que realizan división en complemento a 2 y sin signo, respectivamente. Ellas son:

```
div    $s0, $s1      # Lo=cociente, Hi=resto
divu   $s2, $s3      # versión sin signo de div
```

Para la representación en máquina de las instrucciones precedentes, vea la sección 6.6. Estas instrucciones dejan el cociente y el resto en los registros especiales `Lo` y `Hi`, respectivamente. Observe que, a diferencia de gran parte de la discusión de las secciones 11.4 y 11.5, el dividendo para las instrucciones MiniMIPS `divide` es de ancho sencillo. Por esta razón, está garantizado que el cociente, que siempre es más pequeño en magnitud que el dividendo, encaja en una palabra. Al igual que en el caso de la multiplicación, cuando los resultados se obtengan en `Hi` y `Lo`, se pueden copiar en registros generales, mediante las instrucciones `mfhi` y `mflo`, para ulterior procesamiento.

Por qué, como otras instrucciones de tipo R de MiniMIPS, el cociente y el resto no se colocan en el archivo de registro general, será claro conforme se discuta la ejecución de instrucciones y la secuenciación de control en la parte cuatro del libro. Por ahora sólo se menciona que la razón tiene relación con la latencia mucho más grande de la división, en comparación con las operaciones suma y lógica/corrimiento.

Ejemplo 11.7: Uso de división en programas MiniMIPS Muestre cómo obtener el residuo de z módulo x ($z \bmod x$), donde z y x son enteros con signo de 32 bits en los registros `$s3` y `$s7`, respectivamente. El resultado se debe colocar en el registro `$t2`. Observe que, para números con signo, la operación *residue* (residuo) es diferente del resto en la división; mientras que el signo del resto se define como igual al signo del dividendo z , los residuos por definición siempre son positivos. Suponga que el registro `$t1` está disponible para resultados anulados si fuere necesario.

Solución: El resultado de $z \bmod x$ es un entero positivo s^{pos} que satisface $s^{\text{pos}} < |x|$ y $z = x \times y + s^{\text{pos}}$ para algún entero y . Es fácil ver que s^{pos} es igual que el resto s de la división z/x cuando $s \geq 0$ y se puede obtener como $s + |x|$ cuando $s < 0$.

```
div    $s3,$s7      # resto formado en Hi
mfhi   $t2          # copia resto en $t2
bgez   $t2,done     # resto positivo es el residuo
move   $t1,$s7      # copia x en $t1; esto es |x| si x ≥ 0
bgez   $s7,noneg    # |x| está en $t1; no necesita negación
sub     $t1,$zero,$s7 # pone -x en $t1; esto es |x| si x < 0
noneg: add  $t2,$t2,$t1 # residuo se calcula como s + |x|
done:   ...
```

En las máquinas que no tienen una instrucción `divide` soportada por hardware, la división se puede realizar en software mediante el algoritmo corrimiento-resta que se estudió en la sección 11.4. Es instructivo desarrollar tal programa para MiniMIPS porque ayuda a comprender mejor el algoritmo. En lo que sigue, se considera la división sin signo y se deja al lector desarrollar la versión con signo.

Ejemplo 11.8: División corrimiento-resta de números sin signo Use el algoritmo corrimiento-resta para definir un procedimiento MiniMIPS que realice la división sin signo z/x , con z y x que le pasan en los registros $\$a2$ – $\$a3$ (mitades superior e inferior del entero z de doble ancho) y $\$a0$, respectivamente. Los resultados deben regresarse a $\$v0$ (resto) y $\$v1$ (cociente).

Solución: El procedimiento siguiente, llamado *shsdi* por “división corrimiento-resta” usa tanto instrucciones como seudoinstrucciones. Los registros Hi y Lo , que retienen las mitades superior e inferior del resto parcial z , se representan mediante $\$v0$ y $\$v1$, respectivamente. El divisor x está en $\$a0$ y el cociente y se forma en $\$a1$. El $(k - j)$ -ésimo bit de y se forma en $\$t1$ y luego se suma un corrimiento izquierdo $\$a1$, ello efectivamente inserta el siguiente bit cociente como el LSB de $\$a1$. El registro $\$t1$ también se usa para aislar el MSB de Lo , de modo que se pueda correr hacia Hi durante corrimiento izquierdo. El registro $\$t0$ retiene el MSB de Hi durante corrimiento izquierdo, de modo que se puede usar para elegir el siguiente dígito de cociente. El registro $\$t2$ se usa como contador que se inicializa a 32 y se reduce por 1 en cada iteración hasta que alcanza 0. El uso de registro en este ejemplo se muestra en la figura 11.15 para fácil referencia.

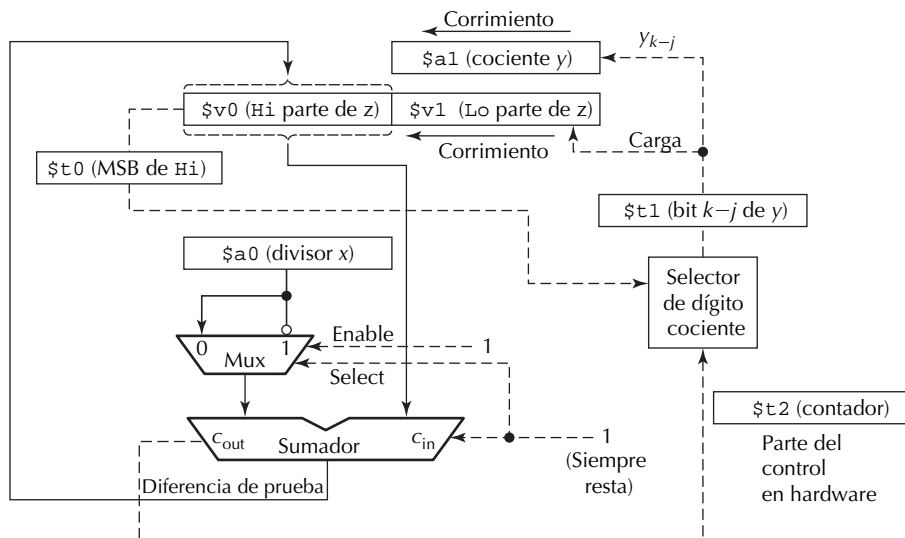


Figura 11.15 Uso de registros para división programada superpuesta en el diagrama de bloques para un divisor de hardware.

```
shsdi: move $v0,$a2      # inicializa Hi a ($a2)
        move $v1,$a3      # inicializa Lo a ($a3)
        addi $t2,$zero,32  # inicializa contador de repetición a 32
dloop: slt  $t0,$v0,$zero  # copia MSB de Hi en $t0
        sll  $v0,$v0,1     # corrimiento izquierdo de parte Hi de z
        slt  $t1,$v1,$zero  # copia MSB de Lo en $t1
        or   $v0,$v0,$t1    # mueve MSB de Lo en LSB de Hi
        sll  $v1,$v1,1     # corrimiento izquierdo de parte Lo de z
        sge  $t1,$v0,$a0    # dígito cociente es 1 si (Hi) ≥ x,
        or   $t1,$t1,$t0    # o si MSB de Hi fue 1 antes de corrimiento
```



```

sll  $a1,$a1,1      # correr y para hacer espacio para nuevo
                    # dígito
or   $a1,$a1,$t1     # copiar y[k-j] en LSB de $a1
beq  $t1,$zero,nosub # si y[k-j] = 0, no restar
subu $v0,$v0,$a0     # restar divisor x de parte Hi de z
nosub: addi $t2,$t2,-1 # reducir contador repetición por 1
bne  $t2,$zero,dloop # si contador > 0, repetir ciclo
                    # división
move $v1,$a1         # copiar el cociente y en $v1
jr   $ra              # regresar al programa llamante

```

Observe que en la sustracción de x de la parte Hi de z , se ignora el bit más significativo del resto parcial que está en $\$t3$. Lo anterior no crea un problema porque la resta se realiza sólo si el resto parcial z (que incluye su MSB oculto) no es menor que el divisor x .

La división entre potencias de 2 se puede hacer mediante corrimiento, que tiende a ser una operación mucho más rápida que la división en cualquier computadora. Así como se puede evitar usar una instrucción *multiply* o rutina de software para ciertos multiplicadores constantes pequeños (último párrafo de la sección 11.3), se puede dividir números por divisores constantes pequeños con el uso de unas cuantas instrucciones *shift/add/subtract*. La teoría para derivar los pasos requeridos es poco más complicada que la usada para la multiplicación y no se discute en este libro. La mayoría de los compiladores modernos son lo suficientemente inteligentes como para reconocer las oportunidades para evitar el uso de una instrucción del tipo *divide* lenta. Las optimizaciones que evitan las operaciones división son incluso más importantes que las que eliminan las multiplicaciones. Esto último sucede porque en los microprocesadores modernos la división es entre cuatro y diez veces más lenta que la multiplicación [Eti02].

PROBLEMAS

11.1 Algoritmo de multiplicación

Dibuja diagramas punto similares al de la figura 11.1 para las siguientes variaciones:

- Multiplicación entera decimal sin signo de 8×4 dígitos. *Sugerencia:* Cambiará el número de puntos.
- Multiplicación fraccional binaria sin signo de 8×4 bits. *Sugerencia:* Cambiará la alineación de los puntos.
- Multiplicación fraccional decimal sin signo de 8×4 dígitos.

11.2 Multiplicación sin signo

Multiplique los siguientes números binarios sin signo de cuatro bits. Presente su trabajo en el formato de la figura 11.2a.

- $x = 1001$ y $y = 0101$
- $x = 1101$ y $y = 1011$
- $x = .1001$ y $y = .0101$

11.3 Multiplicación sin signo

Multiplique los siguientes números decimales sin signo de cuatro bits. Presente su trabajo en el formato de la figura 11.2b.

- $x = 8765$ y $y = 4321$
- $x = 1234$ y $y = 5678$
- $x = .8765$ y $y = .4321$

11.4 Multiplicación en complemento a dos

Represente los siguientes números binarios con magnitud con signo en formato de complemento a 2 de cinco

bits y luego multiplíquelos. Presente su trabajo en el formato de la figura 11.3.

- a) $x = +.1001$ y $y = +.0101$
- b) $x = +.1001$ y $y = -.0101$
- c) $x = -.1001$ y $y = +.0101$
- d) $x = -.1001$ y $y = -.0101$

11.5 Algoritmo de multiplicación

- a) Vuelva a realizar los pasos de multiplicación de la figura 11.2 y verifique que si el producto parcial acumulado se inicializa a 1011 en lugar de 0000, entonces se realizará una operación multiplicar-sumar.
- b) Demuestre que, sin importar el valor inicial de $z^{(0)}$, el resultado multiplicar-sumar con operandos de k dígitos siempre es representable en $2k$ dígitos.

11.6 Arreglo multiplicador

- a) En el arreglo multiplicador de la figura 11.7, etiquete las líneas de entrada con valores de bit correspondientes al ejemplo de multiplicación de la figura 11.2a. Luego determine en el diagrama todos los valores de señal intermedia y de salida y verifique que se obtiene el producto correcto.
- b) Repita la parte a) para $x = 1001$ y $y = 0101$.
- c) Repita la parte a) para $x = 1101$ y $y = 1011$.
- d) ¿Producirá el multiplicador el producto correcto para entradas fraccionales (por ejemplo, $x = 0.1001$ y $y = 0.0101$)?
- e) Muestre cómo se puede realizar la operación multiplicar-sumar en el arreglo multiplicador.
- f) Verifique su método de la parte e) con el uso de los valores de entrada específicos en el problema 11.5a.

11.7 Multiplicación con corrimientos izquierdos

El algoritmo de multiplicación corrimiento-suma que se presentó en la sección 11.1 y se ejemplificó en la figura 11.2, corresponde a hileras de procesamiento de los productos parciales de la matriz de bits (figura 11.1) de arriba abajo. Un algoritmo de multiplicación alternativo en la dirección contraria (abajo arriba) y requiere corrimiento izquierdo del producto parcial antes de cada paso de suma.

- a) Formule este nuevo algoritmo de multiplicación corrimiento-suma en la forma de una ecuación de recurrencia.

- b) Aplique su algoritmo a los ejemplos de la figura 11.2 y verifique que produce las respuestas correctas.
- c) Compare el nuevo algoritmo basado en corrimientos izquierdos con el algoritmo original con corrimientos derechos, en términos de implementación de hardware.

11.8 Multiplicación en MiniMIPS

¿Puede lograr la tarea realizada en el ejemplo 11.3 sin usar otro registro más que $\$t3$ y sin cambiar los registros de operando?

11.9 Multiplicación programada

En la sección 11.2 se afirmó que los registros que retienen el multiplicador y y la mitad inferior del producto parcial acumulado z se pueden combinar en la figura 11.4. ¿Es una combinación similar de los registros $\$a1$ y $\$v1$ (figura 11.8) benéfica en el ejemplo 11.4? Justifique su respuesta mediante la presentación de un procedimiento mejorado o demostrando que la combinación complica el proceso de multiplicación.

11.10 Multiplicación por constantes

Sólo con el uso de instrucciones *shift* y *add/subtract*, diseñe procedimientos eficientes para multiplicación de cada una de las constantes siguientes. Suponga operandos sin signo de 32 bits y asegúrese de que los resultados intermedios no superan el rango de números con signo de 32 bits. No modifique los registros de operando y use sólo otro registro para resultados parciales.

- a) 13
- b) 43
- c) 63
- d) 135

11.11 Multiplicación por constantes

- a) La nueva arquitectura de 64 bits de Intel (IA-64) tiene una instrucción especial *shladd* (corrimiento izquierdo y suma) que puede correr un operando por uno a cuatro bits a la izquierda antes de sumarla con otro operando. Esto último permite que la multiplicación por, digamos, 5, se realice con una instrucción *shladd*. ¿Qué otras multiplicaciones por constantes se pueden realizar mediante una sola instrucción *shladd*?

- b) Repita la parte a) para `shlsub` (corrimiento izquierdo y resta), si supone que el primer operando está con corrimiento.
- c) ¿Cuál es la multiplicación por constante positiva más pequeña para la que se requieren al menos tres de las instrucciones definidas en las partes a) y b)?

11.12 Algoritmo de división

Dibuje diagramas de punto similares al de la figura 11.9 para las variaciones siguientes:

- a) División entera decimal sin signo de 8/4 dígitos. *Sugerencia:* Cambiará el número de puntos.
- b) División fraccional binaria sin signo 8/4 bits. *Sugerencia:* Cambiará la alineación de puntos.
- c) División fraccional decimal sin signo de 8/4 dígitos.

11.13 División sin signo

Realice la división z/x para los siguientes pares dividendo/divisor binario sin signo; obtenga el cociente y y el resto s . Presente su trabajo en el formato de las figuras 11.11 y 11.10.

- a) $z = 0101$ y $x = 1001$
- b) $z = .0101$ y $x = .1001$
- c) $z = 1001\ 0100$ y $x = 1101$
- d) $z = .1001\ 0100$ y $x = .1101$

11.14 División sin signo

Realice la división z/x para los siguientes pares dividendo/divisor decimal sin signo; obtenga el cociente y y el resto s . Presente su trabajo en el formato de las figuras 11.11 y 11.10.

- a) $z = 5678$ y $x = 0103$
- b) $z = .5678$ y $x = .0103$
- c) $z = 1234\ 5678$ y $x = 4321$
- d) $z = .1234\ 5678$ y $x = .4321$

11.15 División con signo

Realice la división z/x para los siguientes pares dividendo/divisor binario con signo; obtenga el cociente y y el resto s . Presente su trabajo en el formato de las figuras 11.11 y 11.10.

- a) $z = +0101$ y $x = -1001$
- b) $z = -.0101$ y $x = -.1001$
- c) $z = +1001\ 0100$ y $x = -1101$
- d) $z = -.1001\ 0100$ y $x = +.1101$

11.16 Arreglo divisor

- a) En el arreglo divisor de la figura 11.14, etiquete las líneas de entrada con los valores de bit correspondiente al ejemplo de división de la figura 11.10a. Luego determine en el diagrama todos los valores de señal intermedios y de salida y verifique que se obtengan el cociente y resto correctos.
- b) Demuestre que las compuertas OR en el borde izquierdo del arreglo divisor de la figura 11.14 se pueden sustituir por celdas MS, lo cual conduce a una estructura más uniforme.
- c) Rote el arreglo divisor 90 grados en contrasentido a las manecillas del reloj, observe que la estructura es similar al arreglo multiplicador y sugiera cómo se pueden combinar los dos circuitos para obtener un circuito que multiplique y divida de acuerdo con el estado de la señal $Mul'Div$.

11.17 Análisis de división de convergencia

Al final de la sección 11.5 se presentó un esquema de convergencia para división con base en refinar iterativamente una aproximación a $1/x$ hasta obtener una buena aproximación $u \cong 1/x$ y luego calcular $q = z/x$ mediante la multiplicación $z \times u$. Demuestre que el método de refinación de la aproximación u que usa la recurrencia $u^{(i+1)} = u^{(i)} \times (2 - u^{(i)} \times x)$ tiene convergencia cuadrática en el sentido de que, si $u^{(i)} = (1 + \epsilon)(1/x)$, entonces $u^{(i+1)} = (1 - \epsilon^2)(1/x)$. Luego comience con la aproximación 0.75 para el recíproco de 1.5, derive aproximaciones sucesivas con base en la recurrencia apenas dada (use aritmética decimal y una calculadora) y verifique que, de hecho, el error se reduce cuadráticamente en cada paso.

11.18 División programada

- a) Describa cómo se puede usar el procedimiento de división programada sin signo del ejemplo 11.8 para realizar una división en la que ambos operandos tengan 32 bits de ancho.
- b) ¿Resultaría alguna simplificación en el procedimiento a partir del conocimiento de que siempre se usará con un dividendo z de un solo ancho (32 bits)?
- c) Modifique el procedimiento del ejemplo 11.8 de modo que realice división con signo.

11.19 División programada

En la sección 11.5 se afirmó que los registros que retienen el cociente y y la mitad inferior del resto parcial z se pueden combinar en la figura 11.12. ¿Una combinación similar de los registros $\$a1$ y $\$v1$ es benéfica en el ejemplo 11.8? Justifique su respuesta mediante la presentación de un procedimiento mejorado o demostrando que la combinación complica el proceso de división.

11.20 División entre 255

Si $z/255 = y$, entonces $y = 256y - z$. Esta observación conduce a un procedimiento para dividir un nú-

mero z entre 255, con el uso de una resta, para obtener cada byte del cociente. Dado que $256y$ termina en ocho 0, el byte menos significativo de y se obtiene con una resta de ancho de byte, para la cual se salva el préstamo. El byte más bajo de y así obtenido es el segundo byte más bajo en $256y$, ello conduce a la determinación del segundo byte más bajo de y mediante otra resta, donde se usa como entrada de préstamo el préstamo salvado. Este proceso continúa hasta que se hayan encontrado todos los bytes de y . Escriba un procedimiento MiniMIPS para implementar este algoritmo de división entre 255 sin alguna instrucción multiplicación o división.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Etie02] Etienne, D., “Computer Arithmetic and Hardware: ‘Off the Shelf’ Microprocessors versus ‘Custom Hardware’”, *Theoretical Computer Science*, vol. 279, núm. 1-2, pp. 3-27, mayo de 2002.
- [Goto02] Goto, G., “Fast Adders and Multipliers”, *The Computer Engineering Handbook*, V. G. Oklobdzija, ed., pp. 9-22 a 9-41, CRC Press, 2002.
- [Knut97] Knuth, D. E., *The Art of Computer Programming*, vol. 2: *Seminumerical Algorithms*, Addison-Wesley, 3a. ed., 1997.
- [Kore93] Koren, I., *Computer Arithmetic Algorithms*, Prentice Hall, 1993.
- [Parh00] Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.
- [Parh02] Parhami, B., “Number Representation and Computer Arithmetic”, *Encyclopedia of Information Systems*, Academic Press, 2002, vol. 3, pp. 317-333.
- [Patt98] Patterson, D. A. y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2a. ed., 1998.
- [Swar90] Swartzlander, E. E., Jr, *Computer Arithmetic*, vols. I y II, IEEE Computer Society Press, 1990.

ARITMÉTICA CON PUNTO FLOTANTE

“La grandeza moral espuria generalmente se liga a cualquier formulación calculada a partir de un gran número de lugares decimales.”

David Berlinski, Del análisis de sistemas, 1976

“Y esta es una aproximación y no una determinación precisa [del valor de π]... pues esta curva no es recta y no se puede determinar, excepto de manera aproximada.”

Abu Jafar Muhammad al-Khwarizmi, Álgebra, ca. 830

TEMAS DEL CAPÍTULO

- 12.1 Modos de redondeo
- 12.2 Valores y excepciones especiales
- 12.3 Suma con punto flotante
- 12.4 Otras operaciones con punto flotante
- 12.5 Instrucciones de punto flotante
- 12.6 Precisión y errores resultantes

Los requerimientos de procesamiento de señales de multimedia y aplicaciones de otros tipos transformaron la aritmética de punto flotante de una herramienta exclusiva para programas científicos y de ingeniería a una capacidad necesaria para todos los usuarios de computadoras. Los atributos principales del formato estándar de punto flotante ANSI/IEEE se presentaron en el capítulo 9. En el presente capítulo se discutirán algunos de los detalles del formato estándar y la manera cómo se realizan las operaciones aritméticas en números con punto flotante. Asimismo, se consideran las operaciones aritméticas básicas (suma, resta, multiplicación, división, raíz cuadrada), así como la evaluación de funciones en general. Además, se revisan conversiones numéricas, redondeo, excepciones y temas relacionados con la precisión y errores de los resultados. También se incluyen instrucciones MiniMIPS para aritmética con punto flotante.

■ 12.1 Modos de redondeo

El formato estándar de punto flotante ANSI/IEEE se describió en términos de sus componentes y atributos principales en la sección 9.6. En ésta y en la siguiente sección, se cubren otras características del estándar, incluso cómo tratar con redondeo y excepciones, como preparación para la discusión acerca de la aritmética con punto flotante en el resto de este capítulo.

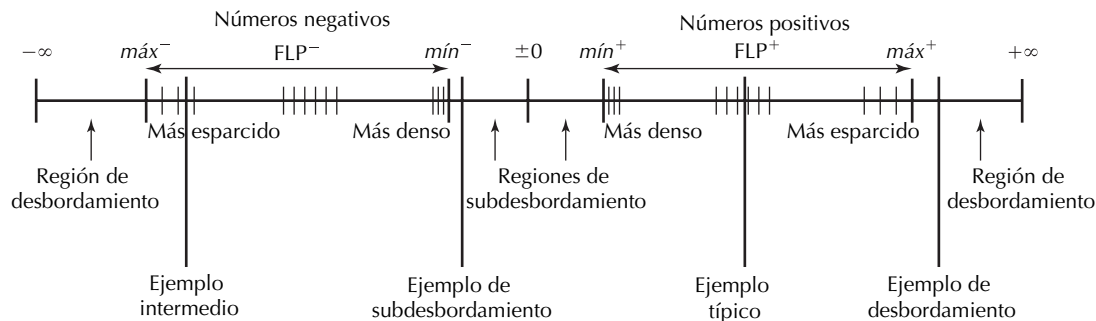


Figura 12.1 Distribución de números con punto flotante en la recta real.

Primero, considere la figura 12.1 que muestra la recta de números reales que se extiende desde $-\infty$ a la izquierda hasta $+\infty$ a la derecha. Los números con punto flotante (excepto los no normales, que se discuten más adelante) se representan mediante marcas cortas en la recta. Los valores especiales ± 0 y $\pm \infty$ se muestran con marcas gruesas, como los números representativos extremos max^+ , max^- , min^+ y min^- que delimitan el rango de números ordinarios con punto flotante positivos (FLP^+) y negativos (FLP^-). Unas cuantas marcas dentro de cada uno de los dos rangos FLP^+ y FLP^- muestran ejemplos de números representativos. Se observa que dichas marcas están más cerca unas de otras alrededor de min^\pm y más separadas cerca de max^\pm . Esto último sucede porque el mismo cambio mínimo de 1 *ulp* en el significando conduce a mayor cambio de magnitud con exponentes más grandes que con exponentes pequeños. En este sentido, la densidad de los números con punto flotante a lo largo de la recta se reduce conforme aumentan sus magnitudes. Ésta es una pista de la utilidad de los números con punto flotante, pues ellos pueden representar números pequeños con buena precisión mientras se sacrifica precisión para números más grandes con el propósito de que se logre mayor rango. Observe que, aun cuando la precisión se reduzca conforme se aleja de 0, el error en la representación (diferencia entre un valor exacto que se necesita representar y la marca más cercana en la figura 12.1) aumenta en términos absolutos, pero permanece casi constante cuando se considera en términos relativos.

A diferencia de los enteros, los números con punto flotante son, por naturaleza, inexactos. Por ejemplo, $1/3$ no tiene una representación exacta en el formato de punto flotante ANSI/IEEE (desde luego, los números reales como π , e y $\sqrt{2}$ tampoco tienen representaciones exactas). Incluso cuando los operandos son exactos, los resultados de las operaciones aritméticas con punto flotante pueden ser inexactos. Lo anterior ocurre, por ejemplo, si se divide 1 entre 3 o se extrae la raíz cuadrada de 2. Tales números corresponden a puntos en la recta real que no coinciden con alguna de las marcas de la figura 12.1. Cuando esto último ocurre, la marca más cercana al valor real se elige como su representación en punto flotante. El proceso de obtener la mejor representación posible en punto flotante para un valor real específico se conoce como *redondeo*.

En la figura 12.1 se muestran cuatro ejemplos de redondeo. Para el caso típico, el número real cae entre dos marcas y está más cerca de una que de otra. Aquí se elige el más cercano de los dos números en punto flotante adyacentes como el valor redondeado para minimizar el error de redondeo. Este esquema de *redondeo al más cercano* también se usa en cálculos manuales. Cuando el número real está en medio de dos marcas, la distancia a ambas es la misma y cualquiera de los dos podría considerarse como la mejor aproximación. En cálculos matemáticos, los casos intermedios siempre se redondean hacia arriba (así, 3.5 se redondea a 4 y 4.5 a 5). El estándar ANSI/IEEE prescribe redondeo al número con punto flotante cuyo significando tiene un LSB de 0 (de dos números con punto flotante adyacentes, el significando de uno debe terminar en 0 y el otro en 1). Este procedimiento de *redondeo al par más*

cercano tiende a resultar en errores negativo y positivo más simétricos y en mayor probabilidad de que los errores se cancelen mutuamente en el curso de un cálculo largo. Observe que un significando que termina en 0 es “par” cuando se estima como entero. Cuando este esquema se aplica para redondear valores a enteros, 3.5 y 4.5 se redondean ambos a 4, el número par más cercano.

Los dos casos descritos cubren la mayoría de las situaciones que ocurren en la práctica. Ocasionalmente se deben representar los números en las regiones de desbordamiento y subdesbordamiento. En estos casos, redondear resulta un poco truculento. Considere desbordamiento positivo (el desbordamiento negativo es similar). En esta región todos los números se redondean a $+\infty$; las únicas excepciones son los valores que están cerca de max^+ que se habrían redondeado hacia abajo si hubiese un siguiente número con punto flotante después de max^+ . De manera similar, en la región de subdesbordamiento positivo sólo los números que están muy cerca de la frontera derecha se redondearían a min^+ . Los otros valores se redondean abajo a $+0$ (se redondearían arriba hacia -0 en la región de subdesbordamiento negativo). Por tanto, $+0$ y -0 , aunque numéricamente equivalente, puede portar información acerca de cómo se produjeron en el curso del cálculo. Esta opción de *alineación a cero* se usa en unidades de punto flotante que no reconocen no normales, la cual se aborda ahora.

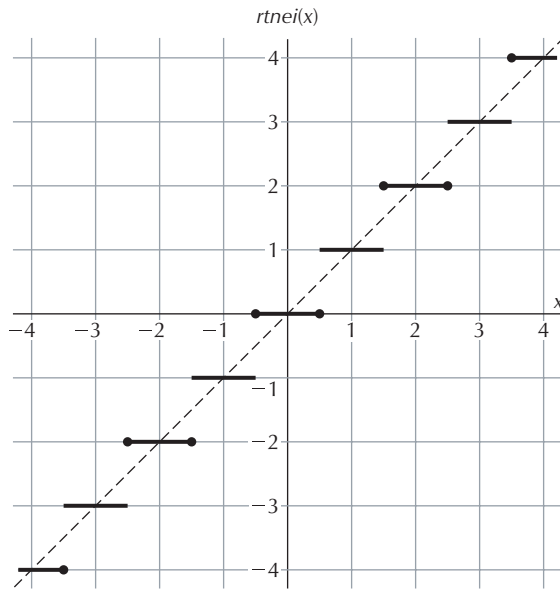
Todos los números con punto flotante ANSI/IEEE se representan en la figura 12.1, excepto para no normales, que cae entre ± 0 y min^\pm . Observe que el ancho de cada una de las dos regiones de la figura 12.1 es 2^{-126} para formato corto y 2^{-1022} para formato largo. Estos son números extremadamente pequeños, aunque si un número que cae en una de las regiones de subdesbordamiento se alinea a cero, el error de representación relativo será bastante significativo. Los no normales suavizan este estallido al permitir más precisamente la representación de valores en estas dos regiones. Por esta razón, a veces se dice que los no normales permiten *subdesbordamiento grácil* o gradual. Con éstos, las regiones de subdesbordamiento en la figura 12.1 se llenan con marcas que están separadas a la misma distancia que las marcas inmediatamente al otro lado de min^\pm . El número más pequeño representable ahora es $2^{-23} \times 2^{-126} = 2^{-149}$ (significando más pequeño posible sin un 1 oculto, multiplicado por la potencia de 2 más baja posible) en formato corto y 2^{-1074} en formato largo, ello hace el subdesbordamiento un problema menos serio.

Ejemplo 12.1: Redondeo al entero más cercano

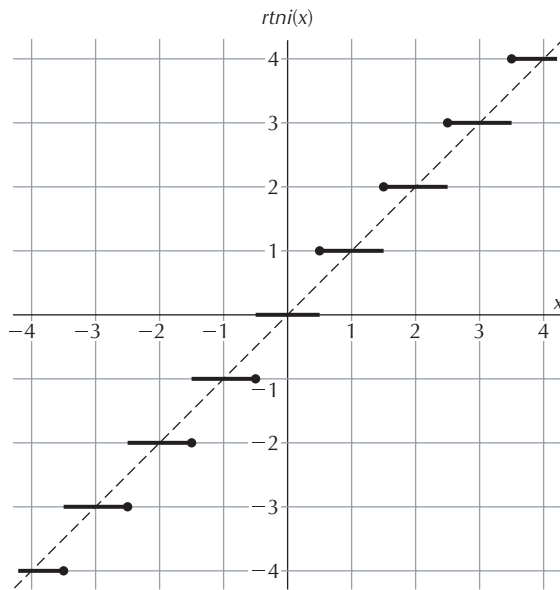
- Considere el entero par redondeado correspondiente a un número real x con magnitud y signo como una función $rnei(x)$. Grafique esta función de redondeo al entero par más cercano para x , en el rango $[-4, 4]$.
- Repita la parte *a*) para la función $rtni(x)$, esto es, la función redondeo al entero más cercano, donde los valores intermedios siempre se redondean hacia arriba.

Solución

- Vea la figura 12.2a. Los puntos gruesos indican cómo se redondean los valores intermedios. Observe que todos los puntos gruesos aparecen en líneas de rejilla horizontal asociados con enteros pares. La línea diagonal punteada representa la función identidad $f(x) = x$ correspondiente a la representación ideal (libre de error) de x . Las desviaciones de esta línea constituyen errores de redondeo.
- Vea la figura 12.2b. La única diferencia con la parte *a*) está en cómo se tratan algunos de los valores intermedios. Por ejemplo, 2.5 se redondea a 3 y -0.5 a -1 .



a) Redondeo al entero par más cercano



b) Redondeo al entero más cercano

Figura 12.2 Dos funciones de redondeo al entero más cercano para x en $[-4, 4]$.

Además del redondeo al par más cercano, que es el modo de redondeo por defecto para el estándar ANSI/IEEE, se definen tres modos de *redondeo dirigido* que permiten al usuario controlar la dirección de los errores de redondeo:

1. *Redondeo hacia adentro (hacia 0)*: Elija el valor más cercano en la misma dirección que 0.
2. *Redondeo hacia arriba (hacia $+\infty$)*: Elija el mayor de los dos valores posibles.
3. *Redondeo hacia abajo (hacia $-\infty$)*: Elija el más pequeño de los dos valores posibles.

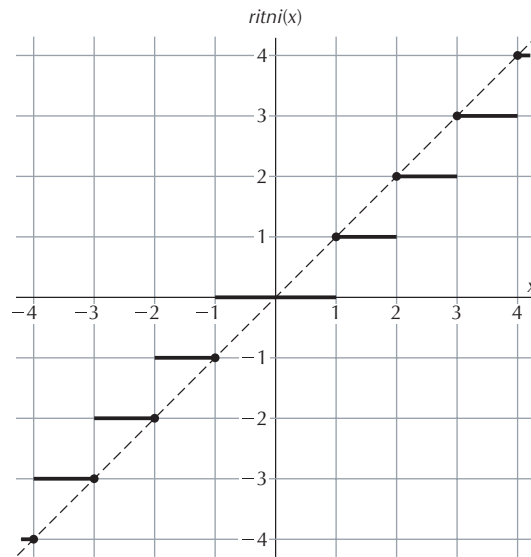
Los modos de redondeo dirigidos hacia adentro, hacia arriba y hacia abajo se citan aquí a manera de una presentación integral. En la mayoría de los cálculos con punto flotante que se procesan, es satisfactorio el modo por defecto de redondeo al par más cercano. Se puede usar redondeo dirigido, por ejemplo, cuando se quiere estar seguro de que un resultado impreciso se garantiza que es menor que el valor correcto (es una cota inferior para el resultado real). Si se necesita calcular z/x de esta forma, donde z y x son resultados de otros cálculos, entonces se debe derivar z con redondeo hacia abajo, x con redondeo hacia arriba y también redondear hacia abajo el resultado de la división con punto flotante.

Ejemplo 12.2: Redondeo dirigido

- a) Considere el entero redondeado dirigido hacia adentro, correspondiente a un número real x con magnitud y signo, como una función $\text{ritni}(x)$. Grafique esta función de redondeo hacia adentro al entero más cercano para x en el rango $[-4, 4]$.
- b) Repita la parte a) para la función $\text{rutni}(x)$ de redondeo hacia arriba al entero más cercano.

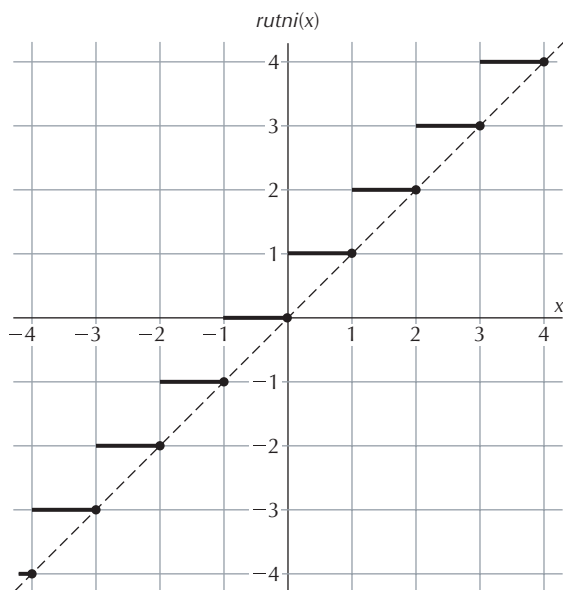
Solución

- a) Vea la figura 12.3a. La línea diagonal punteada es la función identidad $f(x) = x$ correspondiente a la representación ideal (libre de error) de x . Las desviaciones de esta línea representan errores de redondeo.
- b) Observe la figura 12.3b. La diferencia de la parte a) está en los números positivos que se redondean hacia arriba.



a) Redondeo hacia adentro al entero más cercano

Figura 12.3 Dos funciones dirigidas de redondeo al entero más cercano para x en $[-4, 4]$.



b) Redondeo hacia arriba al entero más cercano

Figura 12.3 (Continúa)

12.2 Valores y excepciones especiales

El estándar ANSI/IEEE de punto flotante especifica cinco valores especiales:

± 0	Exponente sesgado = 0, significando = 0 (no 1 oculto)
$\pm \infty$	Exponente sesgado = 255 (corto) o 2047 (largo), significando = 0
NaN	Exponente sesgado = 255 (corto) o 2047 (largo), significando $\neq 0$

En realidad, NaN especifica una clase de valores especiales con el significando no cero potencialmente portando información adicional acerca de lo que se representará. Sin embargo, en este libro no se distingue entre diferentes NaN y se les trata a todas como un solo tipo especial. En un sentido, los no normales también son valores especiales. Sin embargo, después de notar que no tienen un 1 oculto y que su exponente está codificado de manera diferente, se les trata como números ordinarios en punto flotante en el contexto de varias operaciones aritméticas. No se discuten los no normales en esta sección.

Es un poco sorprendente que 0 se considere un valor especial. Sin embargo, esto es consecuencia del uso de un 1 oculto, ello requiere que todos los bits 0 a la cabeza de un significando se eliminen mediante corrimientos izquierdos hasta que su bit 1 de la extrema izquierda se convierte en el 1 oculto. Esto último hace imposible representar un significando de 0 en la forma normal, ya que no tiene 1 en la extrema izquierda para correr y ocultar. En operaciones aritméticas con punto flotante, el 0 se trata en una forma consistente con las reglas de álgebra. Por ejemplo, $x \pm 0 = x$ para cualquier número ordinario x con punto flotante. Si x mismo es un valor especial, entonces las cosas resultan interesantes, como se verá dentro de poco.

También es incomprensible que haya dos tipos de 0. Esto es conveniente de acuerdo con los papeles duales jugados por el 0: el 0 matemático verdadero surge de una operación como $2 - 2$, donde ambos operandos se suponen exactos, y el 0 aproximado que se obtiene al alinear a 0 un resultado subdesbordado. Con el propósito de preservar las identidades matemáticas, $+0$ y -0 se consideran iguales, de modo que la prueba de igualdad $+0 = -0$ produce “verdadero” (*true*) y $-0 < +0$ produce “falso” (*false*) (observe que es tentador considerar el último resultado como “verdadero”). Cuando un resultado x se subdesborda del lado negativo (positivo), se alinea a -0 ($+0$) con el fin de preservar la información de signo. Si, subsecuentemente, se calcula $1/x$ para $x = +0$ o -0 , se obtiene $+\infty$ o $-\infty$, de manera respectiva. Observe que aun cuando pequeños valores negativos y positivos no son muy diferentes y no causa problema alguno considerarlos iguales, $+\infty$ y $-\infty$ son muy diferentes. Así que tener $+0$ y -0 conlleva un propósito útil. Preservar la información de signo requiere que se defina, por ejemplo,

$$(+0) + (+0) = (+0) - (-0) = +0$$

$$(+0) \times (+5) = +0$$

$$(+0)/(-5) = -0$$

Los operandos especiales $+\infty$ y $-\infty$ tienen sus significados matemáticos estándar. Como se observó, 1 dividido entre $+0$ produce $+\infty$ y 1 dividido entre -0 produce $-\infty$, como se esperaba. Igualmente lógicas son las siguientes reglas de muestra, donde x resulta cualquier número de punto flotante no especial:

$$(+\infty) + (+\infty) = +\infty$$

$$x - (+\infty) = -\infty$$

$$(+\infty) \times x = \pm\infty, \text{ dependiendo del signo de } x$$

$$x/(+\infty) = \pm 0, \text{ dependiendo del signo de } x$$

$$\sqrt{+\infty} = +\infty$$

En las pruebas de comparación, $+\infty$ es mayor que cualquier valor (excepto para $+\infty$ y NaN) y $-\infty$ es menor que cualquier valor (distinto de $-\infty$ y NaN).

El último tipo de operando especial, NaN, es útil porque algunas operaciones no producen un valor numérico válido y es conveniente tener una representación especial para tales resultados que de otro modo serían irrepresentables. Por ejemplo, $y = z/x$ es indefinida, en un sentido matemático, cuando $z = x = 0$. Si un programa intenta realizar esta operación, se puede elegir detener su ejecución y producir un mensaje de error para el usuario. Sin embargo, puede resultar que el valor de y nunca se use realmente más tarde en el programa. Como ejemplo, considere el siguiente fragmento de programa, donde los valores actuales de la variable se proporcionan en los comentarios:

$y = z/x$	# actualmente, $z = 0, x = 0$
$v = w/u$	# actualmente, $w = 2, u = 3$
si $a < b$ entonces $c = v$ también $c = y$	# actualmente, $a = 4, b = 5$

Es claro que, aun cuando y es indefinida, el resultado del fragmento de programa en el escenario actual no depende de y ; por tanto, asignar el valor especial NaN a y permite realizar el cálculo y obtener una conclusión exitosa. He aquí algunas otras situaciones que producen NaN como resultado:

$$(+\infty) + (-\infty) = \text{NaN}$$

$$(\pm 0) \times (\pm \infty) = \text{NaN}$$

$$(\pm \infty)/(\pm \infty) = \text{NaN}$$

En operaciones aritméticas, el valor NaN se propaga al resultado. Esto último resulta consistente con la noción intuitiva de un valor indefinido. Por ejemplo:

$$\text{NaN} + x = \text{NaN}$$

$$\text{NaN} + \text{NaN} = \text{NaN}$$

$$\text{NaN} \times 0 = \text{NaN}$$

$$\text{NaN} \times \text{NaN} = \text{NaN}$$

El valor especial NaN no está ordenado con respecto a cualquier otro valor, incluido él mismo. Por tanto, las pruebas de comparación como $\text{NaN} < 2$, $\text{NaN} = \text{NaN}$ o $\text{NaN} \leq +\infty$ regresan “falso”, mientras que $\text{NaN} \neq 2$ o $\text{NaN} \neq \text{NaN}$ regresan “verdadero”.

Ejemplo 12.3: Valores especiales en punto flotante Considere el cálculo $v = z/x + w/x$.

- ¿Cuál es el resultado de este cálculo si actualmente $z = +2$, $w = -1$ y $x = +0$?
- ¿Cuál sería el resultado si se usara la forma algebraicamente equivalente $v = (z + w)/x$?
- ¿Cómo cambiarían los resultados de las partes a) y b) si $x = -0$ en lugar de $+0$?

Solución

- Se obtiene $z/x = +\infty$ y $w/x = -\infty$, ello conduce a $v = (+\infty) + (-\infty) = \text{NaN}$.
- El resultado del cálculo equivalente es $v = (2 - 1)/0 = +\infty$. Observe que este resultado es “mejor” que el de la parte a). Si subsecuentemente se calcula $u = 1/v$, el $+\infty$ que resulta lleva a $1/(+\infty) = 0$, que es consistente con lo que se obtendría del cálculo directo $x/(z + w)$. Con el resultado NaN de la parte a), se obtendría $1/v = 1/\text{NaN} = \text{NaN}$.
- En la parte a) todavía se obtiene $v = \text{NaN}$. En la parte b) el resultado se convierte en $v = -\infty$.

Además de la excepción señalada al comparar valores no ordenados, el estándar ANSI/IEEE también define excepciones asociadas con la división entre cero, desbordamiento, subdesbordamiento, resultado inexacto y operación inválida. Los primeros tres se explican por ellos mismos. La “excepción inexacta” se señala cuando el resultado exacto no redondeado de una operación o conversión no es representativo. La excepción “operación inválida” ocurre en las situaciones siguientes, entre otras:

Suma	$(+\infty) + (-\infty)$
Multiplicación	$0 \times \infty$
División	$0/0$ o ∞/∞
Raíz cuadrada	Operando < 0

En el documento de estándar ANSI/IEEE [IEEE85] se proporciona una descripción más completa. Un grupo de trabajo de expertos discute algunas revisiones y declaraciones al estándar.

12.3 Suma con punto flotante

Sólo se discute la suma con punto flotante porque la resta se puede convertir a suma mediante el cambio de signo del sustraendo. Para realizar suma, los exponentes de los dos operandos en punto flotante, si fuera necesario, se deben igualar. Considere la suma de $\pm 2^{e_1}s_1$ y $\pm 2^{e_2}s_2$, con $e_1 > e_2$. Al hacer ambos exponentes iguales a e_1 , la suma se convierte en:

$$(\pm 2^{e_1}s_1) + (\pm 2^{e_1}(s_2/2^{e_1-e_2})) = \pm 2^{e_1}(s_1 \pm s_2/2^{e_1-e_2})$$

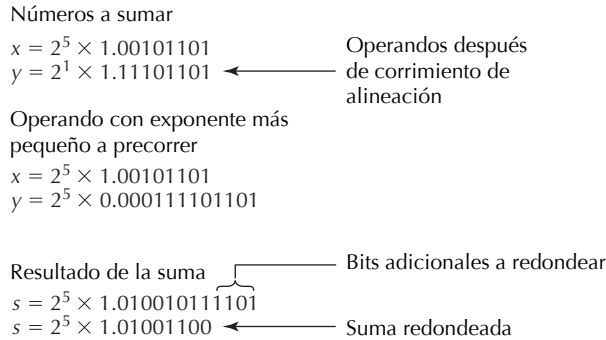


Figura 12.4 Corrimiento de alineación y redondeo en la suma con punto flotante.

Por tanto, se ve que s_2 debe correrse a la derecha por $e_1 - e_2$ bits antes de sumarse a s_1 . Este *corrimiento de alineación* también se llama *precorrimiento* (para distinguirlo del *poscorrimiento* necesario para normalizar un resultado en punto flotante). La figura 12.4 muestra un ejemplo completo de suma con punto flotante, incluso el *precorrimiento*, la suma de significandos alineados y el redondeo final. En este ejemplo, no se necesita *poscorrimiento* porque el resultado ya está normalizado. Sin embargo, en general, el resultado puede necesitar un corrimiento derecho de 1 bit, cuando está en $[2, 4)$, o un corrimiento izquierdo multibit cuando la suma de operandos con diferentes signos conduce a *cancelación* o *pérdida de significancia* y en el resultado aparecen uno o más 0 a la cabeza.

A partir del diagrama de bloques simplificado para un sumador en hardware de punto flotante de la figura 12.5, se ve que, cuando los operandos se desempacan, sus exponentes y significandos se procesan en dos pistas separadas, cuyas funciones están coordinadas por el bloque etiquetado “Control y lógica de signo”. Por ejemplo, con base en la diferencia de los dos exponentes, esta unidad decide cuál operando se debe precorrer. Con el propósito de economizar en hardware, usualmente sólo se proporciona un precorredor, por decir del operando izquierdo del sumador. Si el otro operando necesitara precorrimiento, los operandos se intercambian físicamente. Además, al sumar operandos con signos diferentes, se complementa el operando que no se precorre. Esta estrategia de ahorro de tiempo puede conducir al cálculo de $y - x$ cuando de hecho $x - y$ es el resultado deseado. La unidad de control corrige este problema al forzar la complementación del resultado, si fuera necesario. Finalmente, se realizan la normalización y redondeo, y el exponente se ajusta.

El sumador de significandos de la figura 12.5 puede tener cualquier diseño, pero en la práctica siempre resulta uno de los tipos más rápidos, como el sumador con anticipación de acarreo, discutido en la sección 10.4. El circuito de intercambio consiste de dos multiplexores que permiten que cualquier significando se envíe a cualquier lado del sumador. El *shifter* (corredor) de alineación está diseñado de acuerdo con los métodos discutidos en la sección 10.5 (vea, en particular, la figura 10.17). Como consecuencia de que el corrimiento de alineación siempre es hacia la derecha, el *shifter* resultante será más simple y más rápido que la versión general que se muestra en la figura 10.17. Por ejemplo, cualquier corrimiento de 0 a 31 bits se puede ejecutar en tres etapas:

- 0, 8, 16 o 24 bits en la etapa 1 (mux de cuatro entradas)
- 0, 2, 4 o 6 bits en la etapa 2 (mux de cuatro entradas)
- 0 o 1 bit en la etapa 3 (mux de dos entradas)

Es posible que el normalizador tenga que correr el resultado un bit a la derecha o un número arbitrario de bits a la izquierda. En lo que sigue, por cuestiones de simplicidad, la discusión se limita a una salida positiva del sumador. Cuando la salida del sumador es igual o mayor que 2, se necesita un corrimiento derecho de un bit. En este caso, se garantiza que el resultado sea menor que 4 (¿por qué?), por lo que se requiere un corrimiento derecho de 1 bit para hacer que la parte entera del número, que

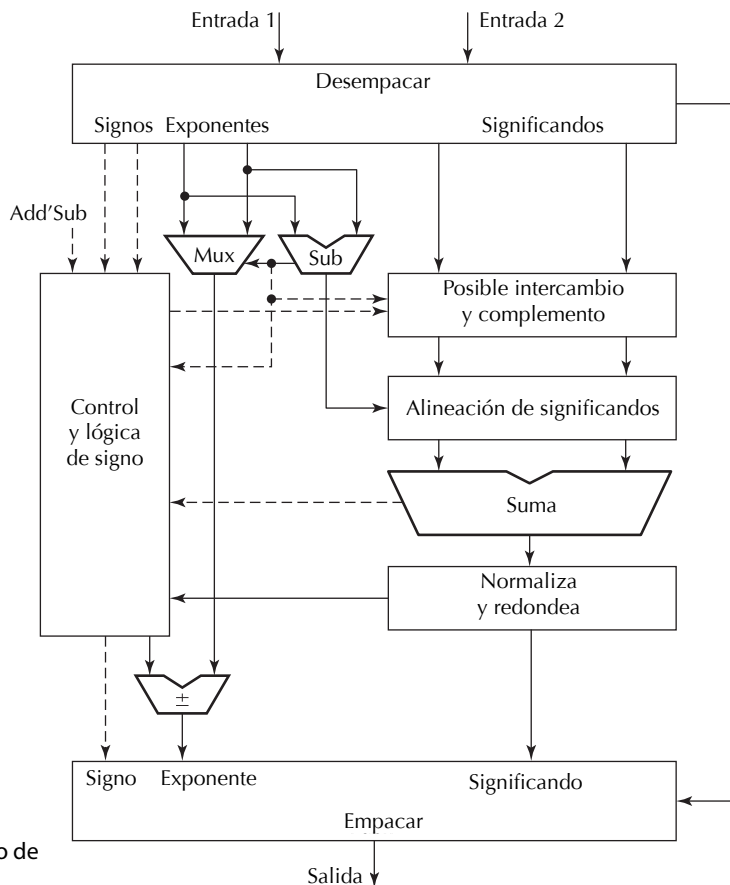


Figura 12.5 Esquema simplificado de un sumador con punto flotante.

es 10 u 11, sea igual a 1; este 1 constituirá entonces el 1 oculto. El corrimiento izquierdo se necesita cuando el resultado tiene uno o más 0 a la cabeza. Pueden existir muchos 0 a la cabeza cuando se suman números de signos diferentes con significandos de valores casi comparables. Si todos los bits del resultado son 0, entonces se requiere un tratamiento especial para codificar el resultado como 0 en la salida. De otro modo se debe contar el número de 0 a la cabeza y el resultado darse al *corrimiento posterior* como su entrada de control.

Finalmente, como se muestra en el ejemplo de la figura 12.4, la salida suma del sumador puede tener bits adicionales como resultado del precorrimiento de un operando. Estos bits se deben descartar y los bits restantes se ajustarán, como corresponde, para un redondeo adecuado. Observe que, en el peor de los casos, el redondeo puede requerir una propagación completa de acarreo a través del ancho del resultado. Por ende, el redondeo resulta un proceso lento que se suma a la latencia de la suma con punto flotante. Por esta razón, y como consecuencia de que en la suma con punto flotante se deben realizar en secuencia muchas operaciones diferentes, se han desarrollado muchos métodos de diseño astutos para hacer operaciones más rápidas. Por ejemplo, el número 0 a la cabeza en el resultado se puede predecir mediante un circuito especial en lugar de determinarse después de que la suma esté disponible. La discusión de estos métodos no es asunto de este libro [Parh00].

Observe que, cuando un significando se normaliza mediante corrimiento, el exponente se debe ajustar en concordancia. Por ejemplo, cuando el significando se corre a la derecha por 1 bit (por tanto,

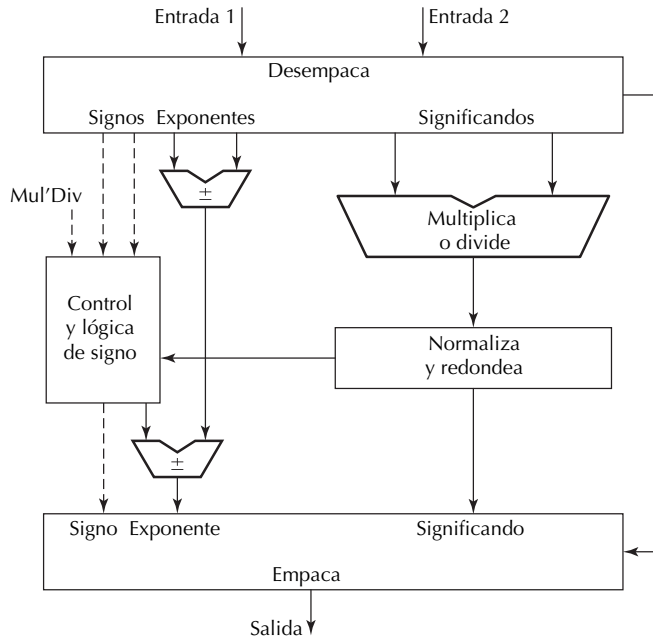


Figura 12.6 Esquema simplificado de una unidad multiplicación/división con punto flotante.

se divide entre 2), el exponente se incrementará por 1 para compensar el cambio. Como resultado de lo anterior, se tiene el pequeño sumador final en la ruta del exponente de la figura 12.5.

12.4 Otras operaciones con punto flotante

Las operaciones de multiplicación y división con punto flotante (figura 12.6) no son muy diferentes de sus contrapartes en punto fijo. Para la multiplicación, los exponentes de los dos operandos se suman y sus significandos se multiplican:

$$(\pm 2^{e_1} s_1) \times (\pm 2^{e_2} s_2) = \pm 2^{e_1+e_2} (s_1 \times s_2)$$

Por ende, un multiplicador en hardware con punto flotante consta de un multiplicador de significando y un sumador de exponente que juntos calculan $2^e s$, con $e = e_1 + e_2$ y $s = s_1 \times s_2$. El signo del resultado se obtiene fácilmente de los signos de los operandos. Sin embargo, esto último no es todo. Con s_1 y s_2 en $[1, 2)$, su producto se encontrará en $[1, 4)$; por tanto, puede estar fuera del rango permitido para un significando. Si el producto de los dos significandos está en $[2, 4)$, dividirlo entre 2 mediante un corrimiento derecho de 1 bit lo colocará de vuelta en el rango deseado. Cuando se requiere esta *normalización*, el exponente e se debe incrementar por 1 para compensar la división entre dos de s .

La división con punto flotante es similar y está regida por la ecuación:

$$(\pm 2^{e_1} s_1) / (\pm 2^{e_2} s_2) = \pm 2^{e_1-e_2} (s_1/s_2)$$

De nuevo, como consecuencia de que la razón s_1/s_2 de los dos significandos se encuentra en $(0.5, 2)$, se puede requerir normalización para los resultados que son menores a 1. Ésta consiste en multiplicar el significando por 2 a través de un corrimiento hacia la izquierda de 1 bit y disminuir el exponente resultante en 1 para compensar la duplicación del significando. A lo largo de la operación y los ajustes consiguientes, tanto para multiplicación como para división, el hardware debe verificar excepciones como *desbordamiento* (exponente muy grande) y *subdesbordamiento* (exponente muy pequeño).

La raíz cuadrada también es muy simple y se puede convertir a la extracción de la raíz cuadrada de s o $2s$ y un simple corrimiento derecho del exponente mediante las reglas siguientes:

$$\begin{aligned}(2^e s)^{1/2} &= 2^{e/2} (s)^{1/2} && \text{cuando } e \text{ es par} \\ &= 2^{(e-1)/2} (2s)^{1/2} && \text{cuando } e \text{ es non}\end{aligned}$$

Observe que tanto $e/2$ para e par y $(e - 1)/2$ para e non se forman mediante un corrimiento derecho de 1 bit de e . Además, tanto $s^{1/2}$ como $(2s)^{1/2}$ ya están normalizados y no se requiere posnormalización (¿por qué?).

Además de las operaciones aritméticas discutidas hasta el momento, son necesarias algunas conversiones para los cálculos con punto flotante. El primer tipo de conversión se refiere a los valores de entrada (usualmente proporcionados en decimal) a la representación interna en punto flotante. Estas conversiones se deben realizar en tal forma que el resultado se redondee de manera adecuada; esto es, el número en punto flotante más cerca posible del valor de entrada real. Para la salida se necesita la conversión inversa, del formato interno en punto flotante a formato decimal.

Otras conversiones que son de interés pertenecen al cambio de un formato en punto flotante a otro (sencillo a doble o viceversa) o entre formatos en punto flotante y entero. Por ejemplo, la conversión de sencillo a doble se necesita cuando las salidas de un programa de precisión sencilla se deben procesar aún más por otro programa que espera entradas de precisión doble. La conversión inversa, de doble a sencillo, tiene algo de “truco” porque requiere redondeo del significando más ancho y verificación de posible desbordamiento.

■ 12.5 Instrucciones de punto flotante

La unidad de punto flotante en MiniMIPS constituye un coprocesador separado (figura 5.1) con sus propios registros y unidad aritmética capaz de realizar operaciones con punto flotante. Existen 32 registros de punto flotante que se denominan del $\$f0$ al $\$f31$. A diferencia de los registros en el archivo de registro general del CPU, no hay convención acerca del uso de los registros de punto flotante, y ningún registro se dedica a retener la constante 0; los 32 registros de punto flotante realmente son unidades de propósito general en este aspecto. Las instrucciones MiniMIPS de punto flotante se dividen en cuatro categorías de operaciones aritméticas, conversiones de formato, transferencias de datos y bifurcaciones condicionales.

Existen diez instrucciones aritméticas de punto flotante que constan de cinco operaciones diferentes (suma, resta, multiplicación, división, negación), cada una con operandos sencillos o dobles. Cada una de dichas instrucciones en lenguaje ensamblador consta de su nombre de operación, seguido por un punto (ignore o lea “flotante”, *float*) y la letra “s” (para sencillo, *single*) o “d” (para doble, *double*). Los operandos sencillos pueden estar en cualquiera de los registros de punto flotante, mientras que los dobles deben especificarse en registros con número par (ocupan el registro especificado y el siguiente en la secuencia, para las mitades superior e inferior del número largo en punto flotante, respectivamente). He aquí algunos ejemplos de instrucciones aritméticas de punto flotante:

```
add.s    $f0,$f8,$f10    # fija $f0 en ($f8) +fp ($f10)
sub.d    $f0,$f8,$f10    # fija $f0 en ($f8) -fp ($f10)
mul.d    $f0,$f8,$f10    # fija $f0 en ($f8) ×fp ($f10)
div.s    $f0,$f8,$f10    # fija $f0 en ($f8) /fp ($f10)
neg.s    $f0,$f8          # fija $f0 en -($f8)
```

La primera instrucción se lee “suma sencillo (flotante)” y la tercera “multiplica doble (flotante)”. La parte “flotante” es opcional en virtud de que “sencillo” y “doble” implican que las operaciones con

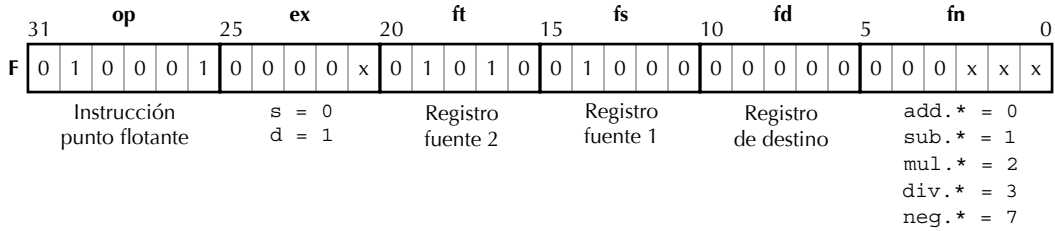


Figura 12.7 Formato común de instrucción de punto flotante para MiniMIPS y los componentes para instrucciones aritméticas. El campo extensión (ex) distingue operandos sencillos (* = s) de dobles (* = d).



Figura 12.8 Instrucciones de punto flotante para conversión de formato en MiniMIPS.

punto flotante se sobreentienden. El formato para estos cinco pares de instrucciones MiniMIPS se muestra en la figura 12.7.

Existen seis instrucciones de conversión de formato: entero a sencillo/doble, sencillo a doble, doble a sencillo y sencillo/doble a entero. Al igual que antes, los operandos sencillos pueden estar en cualquiera de los registros punto flotante, mientras que los dobles deben especificarse en registros con número par. He aquí algunos ejemplos de instrucciones de conversión de formato:

```
cvt.s.w $f0,$f8    # fija $f0 a sencillo(entero $f8)
cvt.d.w $f0,$f8    # fija $f0 a doble(entero $f8)
cvt.d.s $f0,$f8    # fija $f0 a doble($f8)
cvt.s.d $f0,$f8    # fija $f0 a sencillo($f8,$f9)
cvt.w.s $f0,$f8    # fija $f0 a entero($f8)
cvt.w.d $f0,$f8    # fija $f0 a entero($f8,$f9)
```

En estas instrucciones, entero se designa como “w” (para palabra, *word*), mientras que sencillo y doble tienen los símbolos usuales “s” y “d”, respectivamente. Observe que el formato de destino aparece primero, de modo que la primera instrucción anterior se lee “convertir a (flotante) sencillo de entero”. El formato para estas seis instrucciones MiniMIPS se muestra en la figura 12.8.

MiniMIPS tiene seis instrucciones de transferencia de datos: cargar/almacenar palabra a/desde coprocesador 1, mover sencillo/doble desde un registro de punto flotante a otro y mover (copiar) entre registros de punto flotante y registros generales CPU. El uso típico de estas instrucciones es el que sigue:

```
lwc1    $f8,40($s3)    # carga mem[40+($s3)] en $f8
swc1    $f8,A($s3)     # almacena ($f8) en mem[A+($s
mov.s    $f0,$f8        # carga $f0 con ($f8)
mov.d    $f0,$f8        # carga $f0,$f1 con ($f8,$f9)
mfc1    $t0,$f12        # carga $t0 con ($f12)
mtc1    $f8,$t4         # carga $f8 con ($t4)
```

Las primeras dos instrucciones tienen el mismo formato que `lw` y `sw`, que se muestran en la figura 5.7 (es decir, formato I), excepto que el campo `op` contiene 49 para `lwc1` y 50 para `swc1`, y el registro



Figura 12.9 Instrucciones para movimiento de datos de punto flotante en MiniMIPS.

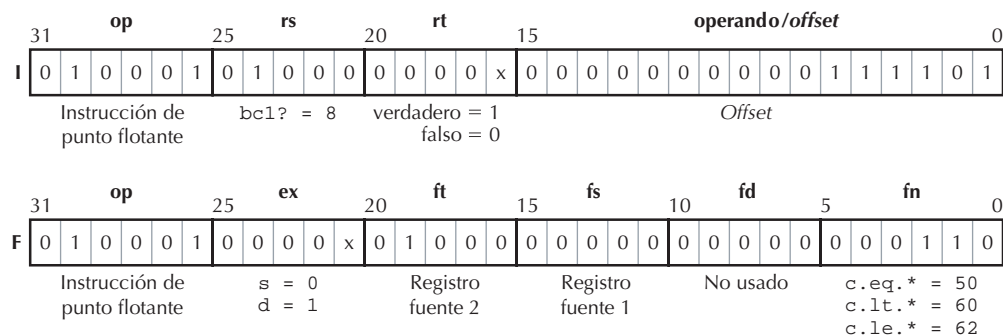


Figura 12.10 Instrucciones de punto flotante bifurcación y comparación en MiniMIPS.

de datos especificado en el campo *rt* representa un registro de punto flotante. Las dos instrucciones que están en medio son instrucciones en formato F que permiten la transferencia de valores punto flotante sencillo o doble de un registro de punto flotante a otro (figura 12.9). Observe que los registros generales no tienen la instrucción *move* (mover) porque su efecto se puede lograr al sumar (\$0) al registro fuente y almacenar el resultado en el registro de destino. Como consecuencia de que la suma con punto flotante constituye una operación mucho más lenta que la suma entera, habría sido inútil seguir la misma estrategia. Finalmente, las últimas dos instrucciones son instrucciones en formato R, como se muestra en la figura 12.9. Estas últimas permiten la transferencia de palabras entre registros de punto flotante y CPU general.

El conjunto final de instrucciones MiniMIPS relacionadas con la unidad de punto flotante consta de dos instrucciones tipo bifurcación y seis de tipo comparación. La unidad de punto flotante tiene una bandera (*flag*) de condición que se establece en *true* o *false* por cualquiera de seis instrucciones comparación: tres tipos de comparación (igual, menor que, menor o igual) para dos tipos de datos (sencillo, doble). Las dos instrucciones *branch on floating-point true/false* (bifurcación en punto flotante verdadero/falso) permiten que la bifurcación condicional se base en el estado de esta bandera.

bc1t	L	# bifurcación en fp flag true
bc1f	L	# bifurcación en fp flag false
c.eq.*	\$f0,\$f8	# si (\$f0)=\$f8, fija flag a "true"
c.lt.*	\$f0,\$f8	# si (\$f0)<(\$f8), fija flag a "true"
c.le.*	\$f0,\$f8	# si (\$f0)≤(\$f8), fija flag a "true"

■ **TABLA 12.1** Las 30 instrucciones MiniMIPS de punto flotante; en virtud de que el campo `op` contiene 17 para todas menos para dos de las instrucciones (49 para `lwc1` y 50 para `swc1`), no se muestra.

Clase	Instrucción	Uso	Significado		
Copiar	Mover registros s/d	<code>mov.* fd, fs</code>	$fd \leftarrow fs$	#	6
	Mover fm coprocesador 1	<code>mfc1 rt, rd</code>	$rt \leftarrow rd$; mueve fp reg a CPU	0	
	Mover a coprocesador 1	<code>mtc1 rd, rt</code>	$rd \leftarrow rt$; mueve CPU reg a fp	4	
Aritmética	Sumar sencillo/doble	<code>add.* fd, fs, ft</code>	$fd \leftarrow (fs) + (ft)$	#	0
	Restar sencillo/doble	<code>sub.* fd, fs, ft</code>	$fd \leftarrow (fs) - (ft)$	#	1
	Multiplicar sencillo/doble	<code>mul.* fd, fs, ft</code>	$fd \leftarrow (fs) \times (ft)$	#	2
	Dividir sencillo/doble	<code>div.* fd, fs, ft</code>	$fd \leftarrow (fs) / (ft)$	#	3
	Negar sencillo/doble	<code>neg.* fd, fs</code>	$fd \leftarrow -(fs)$; cambia bit signo	#	7
	Comparar igual s/d	<code>c.eq.* fs, ft</code>	si $(fs) = (ft)$, fija fp flag a true	#	50
	Comparar menor s/d	<code>c.lt.* fs, ft</code>	si $(fs) < (ft)$, fija fp flag a true	#	60
	Comparar menor o igual s/d	<code>c.le.* fs, ft</code>	si $(fs) \leq (ft)$, fija fp flag a true	#	62
	Convertir entero a sencillo	<code>cvt.s.w fd, fs</code>	$fd \leftarrow sfp(fs)$; (fs) es entero	0	32
Conversión	Convertir entero a doble	<code>cvt.d.w fd, fs</code>	$fd \leftarrow dfp(fs)$; (fs) es entero	0	33
	Convertir sencillo a doble	<code>cvt.d.s fd, fs</code>	$fd \leftarrow dfp(fs)$; fd par	1	33
	Convertir doble a sencillo	<code>cvt.s.d fd, fs</code>	$fd \leftarrow dfp(fs)$; fs par	1	32
	Convertir sencillo a entero	<code>cvt.w.s fd, fs</code>	$fd \leftarrow int(fs)$	0	36
	Convertir doble a entero	<code>cvt.w.d fd, fs</code>	$fd \leftarrow int(fs)$; even fs	1	36
Acceso a memoria	Cargar palabra coprocesador 1	<code>lwc1 ft, imm(rs)</code>	$ft \leftarrow mem[(rs) + imm]$	rs	
	Almacenar palabra coprocesador 1	<code>swc1 ft, imm(rs)</code>	$mem[(rs) + imm] \leftarrow (ft)$	rs	
Transferencia de control	Bifurcación coprocesador 1 true	<code>bc1t L</code>	si (fp flag) = "true", ir a L	8	
	Bifurcación coprocesador 1 false	<code>bc1f L</code>	si (fp flag) = "false", ir a L	8	

* es s/d para sencillo/doble

es 0/1 para sencillo/doble

La figura 12.10 bosqueja la representación en máquina de estas instrucciones. La tabla 12.1 cita todas las instrucciones MiniMIPS de punto flotante cubiertas en esta sección.

Observe que aun cuando se proporcione una instrucción que compara la igualdad de valores en punto flotante, es mejor evitar usarla siempre que sea posible. La razón es que los resultados en punto flotante son inexactos; por ende, si se evalúa la misma función en dos formas diferentes, por decir como $ab + ac$ o $a(b + c)$, se pueden obtener resultados ligeramente diferentes. Esto último se elabora en la sección 12.6.

■ 12.6 Precisión y errores resultantes

Los resultados de los cálculos con punto flotante son inexactos por dos razones. Primero, muchos números no tienen representaciones binarias exactas dentro de un formato de palabra finito. A lo anterior se le refiere como *error de representación*. Segundo, incluso para valores que son exactamente representables, la aritmética de punto flotante produce resultados inexactos. Por ejemplo, el producto exactamente calculado de dos números en punto flotante corto tendrá un significando de 48 bits que se debe redondear para encajar en 23 bits (más el 1 oculto). Lo último se caracteriza como *error de cálculo*. Es importante que tanto los diseñadores de circuitos aritméticos como los usuarios de aritmética de máquina estén atentos a estos errores y aprendan métodos para estimarlos y controlarlos. Existen instancias documentadas de errores aritméticos que conducen a desastres en sistemas cruciales controlados por computadora. Incluso un pequeño error preoperatorio de 0.5 ulp , cuando se compone sobre

Ejemplo 12.4: Violación de las leyes del álgebra en la aritmética con punto flotante En álgebra, las dos expresiones $(a + b) + c$ y $a + (b + c)$ son equivalentes. A esta situación se le conoce como *ley asociativa de la suma*. Mediante la evaluación de estas dos expresiones para los operandos $a = -2^5 \times (1.10101011)_{\text{dos}}$, $b = 2^5 \times (1.10101110)_{\text{dos}}$ y $c = -2^{-2} \times (1.01100101)_{\text{dos}}$, y con las reglas de la aritmética con punto flotante, demuestre que la ley asociativa de la suma no se cumple en general para la aritmética con punto flotante.

Solución: Los pasos del cálculo $(a + b) + c$, y su resultado final $2^{-6} \times (1.10110000)_{\text{dos}}$ se muestran en la figura 12.11a. Asimismo, el cálculo de $a + (b + c)$ conduce al resultado final de 0, como se muestra en la figura 12.11b. El segundo cálculo es menos preciso porque, durante su curso, algunos de los bits de $b + c$ se redondean, mientras que tal pérdida no ocurre en el primer cálculo. Hablando en términos generales, los dígitos significantes se pierden debido a precorrimiento siempre que se suman operandos con diferentes exponentes. Estos dígitos perdidos, que son relativamente poco importantes para la precisión del resultado de dicho paso particular, más tarde pueden volverse cruciales debido a la cancelación de la mayoría o todos los dígitos restantes cuando se realiza una resta con magnitudes de operando comparables.

Números a sumar primero	Números a sumar primero
$a = -2^5 \times 1.10101011$	$b = 2^5 \times 1.10101110$
$b = 2^5 \times 1.10101110$	$c = -2^{-2} \times 1.01100101$
Cálculo de $a + b$	Cálculo de $b + c$ (después de precorrimiento de c)
$2^5 \times 0.00000011$	$2^5 \times 1.101010110011011$
$a+b = 2^{-2} \times 1.10000000$	$b+c = 2^5 \times 1.10101011$ (Redondeo)
$c = -2^{-2} \times 1.01100101$	$a = -2^5 \times 1.10101011$
Cálculo de $(a + b) + c$	Cálculo de $a + (b + c)$
$2^{-2} \times 0.00011011$	$2^5 \times 0.00000000$
Sum = $2^{-6} \times 1.10110000$	Sum = 0 (Normaliza a código especial para 0)
$a) (a +_{\text{fp}} b) +_{\text{fp}} c$	$b) a +_{\text{fp}} (b +_{\text{fp}} c)$

Figura 12.11 Cálculos algebraicamente equivalentes pueden producir resultados diferentes con la aritmética con punto flotante.

muchos millones, quizá miles de millones, de operaciones necesarias en algunas aplicaciones, puede conducir a resultados enormemente imprecisos o totalmente incorrectos.

La discusión de los errores, sus fuentes y contramedidas se limita, en este libro, a unos cuantos ejemplos. Colectivamente, estos ejemplos muestran que, a pesar de sus ventajas obvias, la aritmética con punto flotante puede ser bastante peligrosa y se debe usar con el cuidado adecuado.

Una forma de evitar la acumulación excesiva de errores consiste en acarrear precisión adicional en el curso de un cálculo. Incluso las calculadoras baratas usan dígitos adicionales que son invisibles para el usuario, pero ayudan a garantizar mayor precisión para los resultados. Sin estos *dígitos de protección*, el cálculo de $1/3$ producirá 0.333 333 333 3, si se supone que se trata de una calculadora de diez dígitos. Multiplicar este valor por 3 producirá 0.999 999 999 9, en lugar del esperado 1. En una calculadora con dos dígitos de protección, se evalúa el valor de $1/3$ y se almacena como 0.333 333 333 333, pero todavía se despliega 0.333 333 333 3. Si ahora se multiplica el valor almacenado por 3, y se usa redondeo para derivar el resultado a desplegar, se obtendrá el valor esperado 1.

El uso de los dígitos de protección mejora la exactitud de la aritmética con punto flotante, pero no elimina totalmente algunos resultados incorrectos y sorprendentes, como el incumplimiento de la ley asociativa de la suma discutida en el ejemplo 12.4. Muchas otras leyes del álgebra no satisfacen la aritmética con punto flotante, ello causa dificultades en la predecibilidad y certificación del resultado. ¡Un compilador optimizado que conmute el orden de evaluación por cuestiones de aceleración de cálculo puede cambiar inadvertidamente el resultado obtenido!

Una de las fuentes de dificultades es la pérdida de precisión que ocurre cuando la resta se realiza con operandos de magnitudes comparables. Tal resta produce un resultado que está cerca de 0, lo cual hace bastante significativo, en términos relativos, el efecto de las operaciones de redondeo previo realizadas sobre los operandos. A tal evento se le refiere como *cancelación catastrófica*. Por ejemplo, cuando la ecuación algebraicamente correcta

$$A = [s(s - a)(s - b)(s - c)]^{1/2}$$

con $s = (a + b + c)/2$, se usa para calcular el área de un triángulo con forma de aguja (un triángulo para el que un lado a es casi igual a la suma $b + c$ de los otros dos lados), se puede producir un gran error debido a la cancelación catastrófica en el cálculo $s - a$. Un usuario o programador atento a este problema puede usar una fórmula alterna que no sea proclive para producir tan significativos errores.

Como consecuencia de las anomalías y sorpresas asociadas con la aritmética con punto flotante, existe cierto interés en la *aritmética certificable*. Un ejemplo de lo anterior lo ofrece la *aritmética de intervalo*, mediante la cual cada número se representa por un par de valores, una cota inferior y una cota superior. En este sentido, x se representa mediante el intervalo $[x_l, x_u]$ si se tiene certeza de que $x_l \leq x \leq x_u$. De acuerdo con las representaciones intervalo de x y y , las operaciones aritméticas se definen en tal forma que garanticen contención del resultado en el intervalo que se produce como salida. Por ejemplo:

$$[x_l, x_u] +_{\text{intervalo}} [y_l, y_u] = [x_l +_{\text{fp}\nabla} y_l, x_u +_{\text{fp}\Delta} y_u]$$

Los subíndices “intervalo”, “fp ∇ ” y “fp Δ ” en la expresión anterior califican las operaciones como “suma de intervalo”, “suma con punto flotante con redondeo hacia abajo” y “suma con punto flotante con redondeo hacia arriba”. Éste es un lugar donde los modos de redondeo dirigido abajo y arriba resultan prácticos. Con la aritmética de intervalo, siempre se tiene garantizado un error acotado y se sabrá cuándo un resultado es demasiado impreciso para que sea confiable.

Lo último en certificación de resultado es la *aritmética exacta*, que puede ser factible en algunas aplicaciones mediante el uso de *números racionales* u otras formas de representación exacta. Por ejemplo, si cada valor se representa mediante un signo y un par de enteros para el numerador y denominador, entonces números como $1/3$ tendrán representaciones exactas y una expresión como $(1/3) \times 3$ siempre producirá el resultado exacto. Sin embargo, además de aplicabilidad limitada, la aritmética racional exacta también implica hardware significativo o uso del tiempo del sistema.

En muchos cálculos numéricos, hay necesidad de evaluar funciones como logaritmo, seno o tangente. Un enfoque a la evaluación de funciones es el uso de una *función aproximada*, que es más fácil de evaluar que la función original. Las aproximaciones polinomiales, derivadas de la serie de Taylor y otras expansiones, permiten la evaluación de funciones mediante suma, multiplicación y división. He aquí algunos ejemplos:

$$\ln x = 2(z + z^3/3 + z^5/5 + z^7/7 + \dots) \quad \text{donde } z = (x - 1)/(x + 1)$$

$$e^x = 1 + x/1! + x^2/2! + x^3/3! + x^4/4! + \dots$$

$$\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + x^8/8! - \dots$$

$$\tan^{-1} x = x - x^3/3 + x^5/5 - x^7/7 + x^9/9 - \dots$$

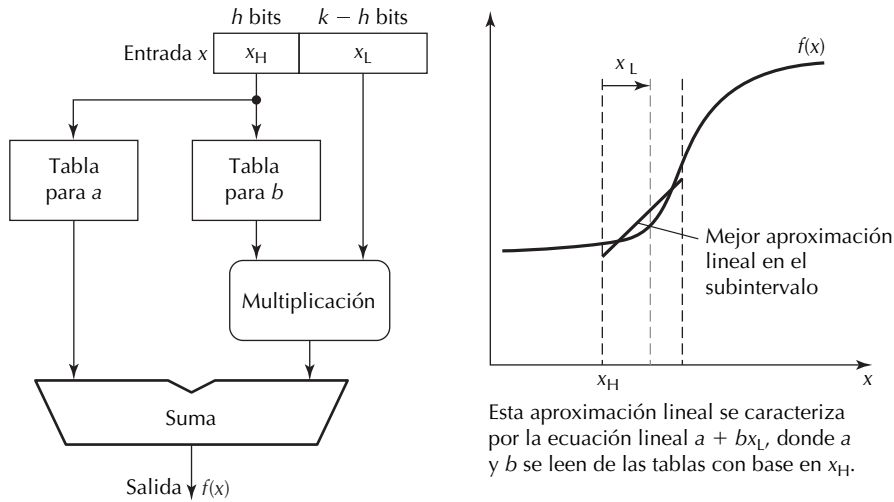


Figura 12.12 Evaluación de función mediante tabla de referencia e interpolación lineal.

Un segundo enfoque es el *cálculo convergente*: comienza con una aproximación adecuada y procede para refinar el valor con evaluación iterativa. Por ejemplo, si $q^{(0)}$ es una aproximación para la raíz cuadrada de x , la siguiente recurrencia se puede usar para refinar el valor, usando una suma, una división y un corrimiento derecho de 1 bit por iteración:

$$q^{(i+1)} = 0.5(q^{(i)} + x/q^{(i)})$$

La aproximación inicial se obtiene mediante la tabla de referencia con base en unos cuantos bits de orden superior de x ; de manera alterna, se puede tomar como una constante.

Tanto con las series de expansión como con el cálculo convergente, se necesitan muchas operaciones en punto flotante para derivar el valor de una función. Este valor tendrá un error que puede ser bastante grande porque, en el peor de los casos, se acumulan los pequeños errores cometidos en varios pasos del cálculo (están en la misma dirección). Tales errores se analizan cuidadosamente y se establecen cotas para ellos, en un esfuerzo por garantizar una precisión razonable para la evaluación de la función como parte de librerías de programa ampliamente disponibles.

Una alternativa al uso de largas secuencias de computación es utilizar tablas de valores precalculados. Entonces estas tablas se consultarán siempre que se necesiten valores de función para argumentos de entrada dados. Desde luego, una tabla que contenga los valores de $\sin x$ para todas las posibles entradas x en punto flotante doble o incluso sencillo, sería imprácticamente grande. Por esta razón, con frecuencia se prefiere un enfoque mixto que involucra tablas mucho más pequeñas y pocas operaciones aritméticas. Este enfoque mixto se basa en el hecho de que, dentro de un intervalo razonablemente estrecho $[x^{(i)}, x^{(i+1)}]$, una función arbitraria $f(x)$ se puede aproximar mediante la función lineal $a + b(x - x^{(i)})$. Este *esquema de interpolación* conduce a la implementación en hardware que se muestra en la figura 12.12. El rango de valores x se divide en 2^h intervalos con base en los h bits de orden superior de x , que definen el valor x_H . Para cada uno de estos intervalos $[x_H, x_H + 2^{-h})$, los correspondientes valores a y b de la función lineal aproximada $a + b(x - x_H) = a + bx_L$ se almacenan en dos tablas. La evaluación de función con este método involucra, por tanto, dos accesos de tabla, una multiplicación y una suma.

PROBLEMAS

12.1 Modos de redondeo

Represente cada uno de los valores $1/3$, $1/4$ y $1/5$ en el formato corto ANSI/IEEE. Redondee valores inexac-

- a) Al valor más cercano
- b) Al par más cercano
- c) Hacia adentro
- d) Hacia abajo
- e) Hacia arriba

12.2 Modos de redondeo

Este problema es continuación del ejemplo 12.1.

- a) Dibuje nuevas versiones de los diagramas de la figura 12.2 si los valores en el rango $[-4, 4]$ se deben redondear a formato binario de punto fijo con un bit fraccional.
- b) Grafique la función $rtoi(x)$ de redondeo al entero no más cercano para x en $[-4, 4]$.
- c) Demuestre que $rtoi(x)$ es más fácil de implementar en hardware que $rtni(x)$ en el sentido de que sólo se afecte LSB durante el proceso de redondeo.
- d) La interferencia, o redondeo von Neumann, se realiza al desechar cualquier bit adicional y forzar el LSB de la parte restante a 1. El proceso casi es tan sencillo como un troceado simple. Compare este modo de redondeo con el troceado y los de las partes a) a c) en términos de los errores introducidos.

12.3 Modos de redondeo

Este problema es continuación del ejemplo 12.2.

- a) Dibuje nuevas versiones de los diagramas de la figura 12.3 si valores en el rango $[-4, 4]$ se deben redondear a formato binario de punto fijo con un bit fraccional.
- b) Grafique la función $rdtni(x)$ de redondeo hacia abajo al entero más cercano para x en $[-4, 4]$.
- c) Compare estos tres modos de redondeo dirigido con los modos de troceado simple y redondeo a más cercano en términos de los errores introducidos.

12.4 Operaciones sobre valores especiales

Para cada una de las siguientes operaciones con punto flotante, dibuje una tabla 5×5 , etiquete las hileras y co-

lumnas con los valores especiales $-\infty$, -0 , $+0$, $+\infty$, NaN. En cada cuadro, escriba el resultado de la operación específica cuando se aplica a los operandos, y etiquete las correspondientes hileras y columnas. Por ejemplo, para el caso de $+_{fp}$, la entrada para la hilera $+\infty$ y la columna $-\infty$ debe ser NaN.

- a) Suma
- b) Resta
- c) Multiplicación
- d) División

12.5 Operaciones con punto flotante

Muestre los resultados de las siguientes operaciones con punto flotante. Justifique sus respuestas.

- a) $\min +_{fp} \max$
- b) $\min -_{fp} \max$
- c) $\min \times_{fp} \max$
- d) $\min /_{fp} \max$
- e) $(\min)^{1/2}$

12.6 Operaciones con punto flotante

Represente cada uno de los operandos decimales en las siguientes expresiones en el formato corto de punto flotante ANSI/IEEE. Luego, realice la operación y derive el resultado usando modo de redondeo al par más cercano.

- a) $1.5 +_{fp} 2^{-23}$
- b) $1.0 -_{fp} (1.5 \times 2^{-23})$
- c) $(1 + 2^{-11}) \times_{fp} (1 + 2^{-12})$
- d) $2^5 /_{fp} (2 - 2^{-23})$
- e) $(1 + 2^{-23})^{1/2}$

12.7 Formato en punto flotante toy

Considere un formato en punto flotante toy de ocho bits con un bit de signo, un campo exponente de cuatro bits y un campo significando de tres bits. Los exponentes 0000 y 1111 se reservan para valores especiales, y el resto se usa para codificar los exponentes -7 (0001) a $+6$ (1110). La base de exponente es 2. El significando tiene un 1 oculto a la izquierda del punto base, con el campo de tres bits que constituye su parte fraccional.

- ¿Cuál es el sesgo de exponente en el formato de punto flotante definido?
- Determine los valores representados más grande (*max*) y más pequeño distinto de cero (*min*).
- Con la exclusión de valores especiales, ¿cuántos valores diferentes se representan con este formato?
- Represente los números $x = 0.5$ y $y = -2$ en este formato.
- Calcule la suma $x + y$ con las reglas de la aritmética con punto flotante. Muestre todos los pasos.

12.8 Operaciones con punto flotante

Represente cada uno de los operandos decimales en las siguientes expresiones en el formato corto de punto flotante ANSI/IEEE. Luego realice la operación y derive el resultado en el mismo formato, y normalice si fuera necesario.

- $(+41.0 \times 2^{+0}) \times_{\text{fp}} (+0.875 \times 2^{-16})$
- $(-4.5 \times 2^{-1}) /_{\text{fp}} (+0.0625 \times 2^{+12})$
- $(+1.125 \times 2^{+11})^{1/2}$
- $(+1.25 \times 2^{-10}) +_{\text{fp}} (+0.5 \times 2^{+11})$
- $(-1.5 \times 2^{-11}) -_{\text{fp}} (+0.0625 \times 2^{-10})$

12.9 Representaciones en punto flotante

Considere el formato corto de punto flotante ANSI/IEEE.

- Si se ignoran $\pm\infty$, ± 0 , NaN y no normales, ¿cuántos números reales distintos son representables?
- ¿Cuál es el número mínimo de bits necesarios para representar todos estos valores distintos? ¿Cuál es la eficiencia de codificación o representación de este formato?
- Discuta las consecuencias (en términos de rango y precisión) de recortar el campo exponente por dos bits y sumar dos bits al campo significando.
- Repita la parte d), pero esta vez suponga que la base del exponente se aumenta de 2 a 16.

12.10 Punto fijo y flotante

Encuentre el valor más grande de n para el cual $n!$ se puede representar en los siguientes formatos de 32 bits. Explique el resultado contrario a la intuición.

- Formato entero en complemento a dos.
- El formato corto ANSI/IEEE.

12.11 Excepciones de punto flotante

- Proporcione ejemplos de números x y y en formato corto ANSI/IEEE tales que lleven a desbordamiento en la etapa de redondeo de la operación $x +_{\text{fp}} y$.
- Repita la parte a) para $x \times_{\text{fp}} y$.
- Demuestre que el redondeo del desbordamiento es imposible en la fase de normalización de la división en punto flotante.

12.12 Redondeo al par más cercano

Este ejemplo muestra una ventaja del redondeo al par más cercano sobre el redondeo ordinario. Todos los números son decimales. Considere los números en punto flotante $u = .100 \times 10^0$ y $v = -.555 \times 10^{-1}$. Comience con $u^{(0)} = u$ y use la recurrencia $u^{(i+1)} = (u^{(i)} -_{\text{fp}} v) +_{\text{fp}} v$ para calcular $u^{(1)}$, $u^{(2)}$, etcétera.

- Demuestre que, con redondeo ordinario, los valores sucesivos de u serán diferentes, una ocurrencia conocida como *deriva*.
- Verifique que la deriva no ocurre en el ejemplo si se redondea al par más cercano.

12.13 Redondeo doble

Considere la multiplicación de valores decimales de dos dígitos y precisión sencilla 0.34 y 0.78, que producen 0.2652. Si este resultado exacto se redondea a un formato interno de tres dígitos, se obtendrá 0.265, que, cuando se redondea subsecuentemente a precisión sencilla mediante redondeo al par más cercano, produce 0.26. Sin embargo, si el resultado exacto se redondea directamente a precisión sencilla, produciría 0.27. ¿El redondeo doble conduce a un problema similar si siempre se redondean los casos intermedios en lugar de usar el modo de redondeo al par más cercano?

12.14 Redondeo de números ternarios

Si en lugar de computadoras binarias se tienen ternarias, la aritmética en base 3 sería de uso común en la actualidad. Discuta los efectos que tal cambio tendría sobre el redondeo en la aritmética de punto flotante. *Sugerencia:* ¿Qué ocurre a los casos intermedios, o a números que están igualmente espaciados de los números con punto flotante sobre cualquier lado?

12.15 Errores de cálculo

Considere la secuencia $\{u^{(i)}\}$ definida por la recurrencia $u^{(i+1)} = i \times u^{(i)} - i$, con $u^{(1)} = e$.

- Use una calculadora o escriba un programa para determinar los valores de $u^{(i)}$ para i en $[1, 25]$.
- Repita la parte *a)* con una calculadora diferente o con una precisión diferente en su programa.
- Explique los resultados.

12.16 Aritmética de intervalo

Se tienen los números decimales en punto flotante $u = .100 \times 10^0$ y $v = -.555 \times 10^{-1}$.

- Use la aritmética de intervalo para calcular la media de x y y mediante la expresión $(x +_{\text{fp}} y) /_{\text{fp}} 2$.
- Repita la parte *a)*, pero use la expresión $x +_{\text{fp}} [(y -_{\text{fp}} x) /_{\text{fp}} 2]$. Explique cualquier diferencia con la parte *a)*.
- Use la aritmética de intervalo para calcular la expresión $(x \times_{\text{fp}} x) -_{\text{fp}} (y \times_{\text{fp}} y)$.
- Repita la parte *c)*, pero use la expresión $(x -_{\text{fp}} y) \times_{\text{fp}} (x +_{\text{fp}} y)$. Explique cualquier diferencia con la parte *c)*.

12.17 Rendimiento de punto flotante

En el capítulo 4, se sugirió MFLOPS como una medida del rendimiento de la computadora para aplicacio-

nes intensas de punto flotante. En esta parte del libro se aprendió que algunas operaciones aritméticas son intrínsecamente más difíciles y, por tanto, más lentas, que otras. Por ejemplo, dos máquinas que muestran el mismo rendimiento en MFLOPS en dos aplicaciones diferentes pueden tener potencias computacionales sustancialmente diferentes si una aplicación realiza sólo sumas y multiplicaciones en punto flotante mientras que la otra tiene una fuerte dosis de divisiones, raíces cuadradas y exponenciación. Para hacer la comparación más justa, se ha sugerido que cada suma/resta (incluida comparación) o multiplicación se cuenta como una operación, cada división o raíz cuadrada como cuatro operaciones y cada exponenciación u otra evaluación de función como ocho operaciones. Para una aplicación que ejecute 200 mil millones de operaciones en punto flotante en 100 segundos, con el número de diferentes operaciones, en miles de millones, dado dentro de paréntesis, calcule la tasa MFLOPS sin y con las ponderaciones sugeridas. La mezcla de operaciones es cargar (77), almacenar (23), copiar (4), sumar (41), restar (21), multiplicar (32), dividir (1), otras funciones (1). Establezca todas sus suposiciones claramente.

REFERENCIAS Y LECTURAS SUGERIDAS

- | | |
|----------|---|
| [Etie02] | Etienne, D., "Computer Arithmetic and Hardware: 'Off the Shelf' Microprocessors versus 'Custom Hardware'", <i>Theoretical Computer Science</i> , vol. 279, núms. 1-2, pp. 3-27, mayo de 2002. |
| [Gold91] | Goldberg, D., "What Every Computer Scientist Should Know About Floating-Point Arithmetic", <i>ACM Computing Surveys</i> , vol. 23, núm. 1, pp. 5-48, marzo de 1991. |
| [IEEE85] | ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic, disponible en IEEE Press. |
| [Knut97] | Knuth, D. E., <i>The Art of Computer Programming</i> , vol. 2: <i>Seminumerical Algorithms</i> , Addison-Wesley, 3a. ed., 1997. |
| [Parh00] | Parhami, B., <i>Computer Arithmetic: Algorithms and Hardware Designs</i> , Oxford University Press, 2000. |
| [Parh02] | Parhami, B., "Number Representation and Computer Arithmetic", <i>Encyclopedia of Information Systems</i> , Academic Press, 2002, vol. 3, pp. 317-333. |
| [Patt98] | Patterson, D. A. y J. L. Hennessy, <i>Computer Organization and Design: The Hardware/Software Interface</i> , Morgan Kaufmann, 2a. ed., 1998. |
| [Ster74] | Sterbenz, P. H., <i>Floating-Point Computation</i> , Prentice-Hall, 1974. |
| [Swar90] | Swartzlander, E. E., Jr, <i>Computer Arithmetic</i> , vols. I y II, IEEE Computer Society Press, 1990. |

PARTE CUATRO

ruta de datos y control

“La verdad es un río que siempre se divide en brazos que se reúnen. Viviendo entre los brazos, los habitantes debaten toda la vida cuál es el río principal.”

Cyril Connolly

“Las computadoras pueden resolver todo tipo de problemas, excepto las cosas en el mundo que no se pueden calcular.”

Anónimo

TEMAS DE ESTA PARTE

- 13.** Pasos de ejecución de instrucciones
- 14.** Síntesis de unidad de control
- 15.** Rutas encauzadas de datos
- 16.** Límites del rendimiento encauzado

La ruta de datos constituye la parte de un CPU mediante la cual fluyen las señales de datos conforme se manipulan de acuerdo con la definición de la instrucción (por ejemplo, desde dos registros, a través de la ALU y de vuelta a un registro). Las computadoras más simples tienen una ruta de datos lineal que se usa para todas las instrucciones, donde las señales de control determinan la acción, o la falta de ella, en cada parte de la ruta. Esta uniformidad simplifica el diseño e implementación del hardware. Las computadoras más avanzadas tienen rutas de datos que se bifurcan; por tanto, permiten la ejecución concurrente de instrucciones en diferentes partes de la ruta de datos. La unidad de control es responsable de guiar las señales de datos a lo largo de la ruta de datos, e indica a cada parte el tipo de transformación, para el caso de que existiera alguna, que debe realizar en los elementos de datos conforme éstos se mueven.

Después de elegir un pequeño subconjunto de instrucciones MiniMIPS para hacer manejable la discusión y los diagramas de implementación de hardware, el capítulo 13 sigue una ruta de datos simple y un control de ciclo sencillo asociado. En el capítulo 14 se muestra que el control multiciclo permite más flexibilidad, así como potencialmente en mayor rendimiento y eficiencia. En este contexto, todos los procesadores modernos usan encauzamiento (*pipelining*) con el propósito de lograr mayor rendimiento. Este tema se discute en la segunda mitad de la parte cuatro, donde los fundamentos de encauzamiento se cubren en el capítulo 15 y las complicaciones (junto con soluciones para evitar pérdida de rendimiento) se presentan en el capítulo 16.

PASOS DE EJECUCIÓN DE INSTRUCCIONES

“En una computadora mecánica [el encabezado de las instrucciones administrativas] habría sido muy obvia sólo para el curioso y si Babbage hubiese acertado al principio de programa almacenado, indudablemente lo habría rechazado por dicha razón. Con la electrónica, lo que el ojo no ve el corazón no lo llorará.”

Maurice Wilkes, Perspectivas de la computación

“Para entrar a la sala de operaciones, los pacientes que asisten al área de urgencias deben seguir estos pasos: 1) Pasar su tarjeta de crédito, 2) elegir la operación del menú, 3) esperar la autorización.”

Anónimo, Señal ficticia de hospital

TEMAS DEL CAPÍTULO

- 13.1** Un pequeño conjunto de instrucciones
- 13.2** La unidad de ejecución de instrucciones
- 13.3** Una ruta de datos de ciclo sencillo
- 13.4** Bifurcación y saltos
- 13.5** Derivación de las señales de control
- 13.6** Rendimiento del diseño de ciclo sencillo

Las computadoras digitales más simples ejecutan las instrucciones una tras otra, siguiendo un flujo de control desde una instrucción concluida hasta la siguiente en la secuencia, a menos que de manera explícita se dirija para alterar este flujo (por una instrucción bifurcación [*branch*] o salto [*jump*]) o para terminar la ejecución de instrucciones. Por ende, tal computadora se puede ver como si estuviera en un ciclo, donde cada iteración conduce a la conclusión de una instrucción. Parte del proceso de ejecución de instrucciones, que comienza con *fetching* (leer) la instrucción de la memoria en la dirección específica en el contador del programa (PC), consiste en determinar dónde se ubica la siguiente instrucción, así como actualizar el contador de programa como corresponde. En este capítulo se examinan los pasos de ejecución de instrucciones en una computadora sencilla. Más tarde se verá que, para lograr mayor rendimiento, este enfoque se debe modificar de manera sustancial.

13.1 Un pequeño conjunto de instrucciones

Los diseños de ruta de datos CPU y unidad de control que se presentan en los capítulos 13 a 16 se basan en la arquitectura de conjunto de instrucciones MiniMIPS introducidos en la parte dos. Con el propósito de hacer manejable el problema de diseño y los diagramas y tablas menos confusos, las realizaciones de hardware tienen como fundamento una versión de 22 instrucciones de MiniMIPS, que se denomina “MicroMIPS”. El conjunto de instrucciones MicroMIPS es idéntico al que se muestra en la tabla 5.1 al final del capítulo 5, con la única diferencia de la inclusión de las instrucciones `jal` y `syscall` de la tabla 6.2. El último par de instrucciones convierte a MicroMIPS en una computadora completa que puede correr programas sencillos, aunque útiles. Para hacer a este capítulo autocontenido y también de fácil referencia, en la tabla 13.1 se presenta el conjunto de instrucciones MicroMIPS completo. El manejo de las otras instrucciones MiniMIPS, dadas en la tabla 6.2 al final del capítulo 6, forma los temas de muchos problemas de fin de capítulo.

También para referencia, en la figura 13.1 se reproduce una versión más compacta de la figura 5.4, que muestra los formatos de instrucción R, I y J, así como sus diversos campos. Recuerde que, en las instrucciones aritméticas y lógicas con dos operandos de fuente de registro, `rd` especifica el registro de destino. Para instrucciones de tipo ALU con un operando inmediato o para la instrucción *load word* (cargar palabra), el campo `rd` se vuelve parte del operando inmediato de 16 bits. En este caso, `rt` designa el registro de destino. Observe que, de acuerdo con MicroMIPS no hay instrucción *shift* (corrimiento), el campo `sh` (cantidad de corrimiento) no se usa.

TABLA 13.1 Conjunto de instrucciones MicroMIPS.*

Clase	Instrucción	Uso	Significado		
Copiar	Cargar superior inmediato	<code>lui rt, imm</code>	$rt \leftarrow (imm, 0x0000)$	15	
	Suma	<code>add rd, rs, rt</code>	$rd \leftarrow (rs) + (rt)$	0	32
	Resta	<code>sub rd, rs, rt</code>	$rd \leftarrow (rs) - (rt)$	0	34
	Fijar menor que	<code>slt rd, rs, rt</code>	$rd \leftarrow \text{si } (rs) < (rt) \text{ entonces } 1 \text{ sino } 0$	0	42
	Suma inmediata	<code>addi rt, rs, imm</code>	$rt \leftarrow (rs) + imm$	8	
Aritmética	Fijar menor que inmediato	<code>slti rt, rs, imm</code>	$rt \leftarrow \text{si}(rs) < imm \text{ entonces } 1 \text{ sino } 0$	10	
	AND	<code>and rd, rs, rt</code>	$rd \leftarrow (rs) \wedge (rt)$	0	36
	OR	<code>or rd, rs, rt</code>	$rd \leftarrow (rs) \vee (rt)$	0	37
	XOR	<code>xor rd, rs, rt</code>	$rd \leftarrow (rs) \oplus (rt)$	0	38
	NOR	<code>nor rd, rs, rt</code>	$rd \leftarrow ((rs) \vee (rt))'$	0	39
Lógica	AND inmediata	<code>andi rt, rs, imm</code>	$rt \leftarrow (rs) \wedge imm$	12	
	OR inmediata	<code>ori rt, rs, imm</code>	$rt \leftarrow (rs) \vee imm$	13	
	XOR inmediata	<code>xori rt, rs, imm</code>	$rt \leftarrow (rs) \oplus imm$	14	
	Cargar palabra	<code>lw rt, imm(rs)</code>	$rt \leftarrow \text{mem}[(rs) + imm]$	35	
	Almacenar palabra	<code>sw rt, imm(rs)</code>	$\text{mem}[(rs) + imm] \leftarrow (rt)$	43	
Acceso a memoria	Saltar	<code>j L</code>	ir a L	2	
	Saltar registro	<code>jr rs</code>	ir a (rs)	0	8
	Bifurcación en menor que 0	<code>bltz rs, L</code>	si (rs) < 0 entonces ir a L	1	
	Bifurcación en igual	<code>beq rs, rt, L</code>	si (rs) = (rt) entonces ir a L	4	
	Bifurcación en no igual	<code>bne rs, rt, L</code>	si (rs) ≠ (rt) entonces ir a L	5	
	Saltar y ligar	<code>jal L</code>	ir a L; $\$31 \leftarrow (PC) + 4$	3	
	Petición de sistema	<code>syscall</code>	Vea sección 7.6 (tabla 7.2)	0	12

*Nota: Excepto por las instrucciones `jal` y `syscall` en el fondo, que convierten a MicroMIPS en una computadora completa, esta tabla es la misma que la tabla 5.1.

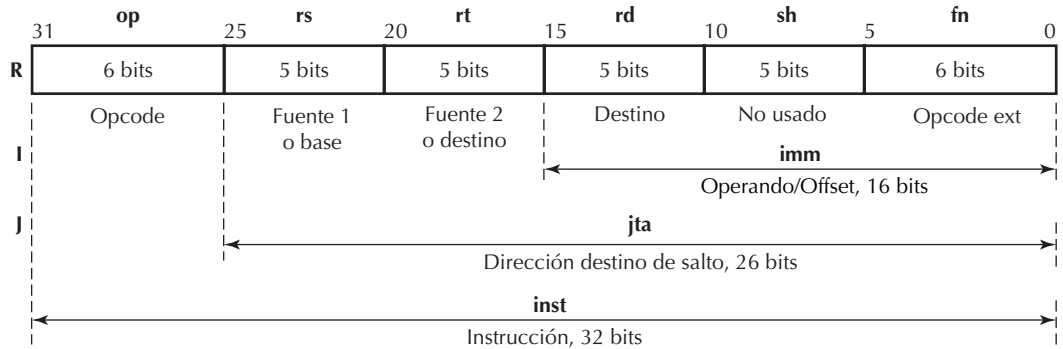


Figura 13.1 Formatos de instrucción MicroMIPS y nombre de los diversos campos.

Las instrucciones de la tabla 13.1 se pueden dividir en cinco categorías con respecto a los pasos que son necesarios para su ejecución:

- Siete instrucciones ALU formato R (add, sub, slt, and, or, xor, nor)
- Seis instrucciones ALU formato I (lui, addi, slti, andi, ori, xori)
- Dos instrucciones de acceso a memoria formato I (lw, sw)
- Tres instrucciones de bifurcación condicional formato I (bltz, beq, bne)
- Cuatro instrucciones de salto incondicional (j, jr, jal, syscall)

Las siete instrucciones ALU con formato R tienen la siguiente secuencia de ejecución común:

1. Leer los contenidos de los registros fuente *rs* y *rt*, y adelantarlos como entradas a la ALU.
2. Decir a la ALU qué tipo de operación realizar.
3. Escribir la salida de la ALU en el registro destino *rd*.

Cinco de las seis instrucciones ALU con formato I requieren pasos similares a los tres mencionados, excepto que los contenidos de *rs* y el valor inmediato en la instrucción se adelantan como entradas a la ALU y el resultado se almacena en *rt* (en lugar de en *rd*). La única excepción es *lui*, que sólo requiere el operando inmediato, pero, incluso en este caso, leer el contenido de *rs* no causará daño, ya que la ALU puede ignorarlo. La discusión anterior abarca 13 de las 21 instrucciones MicroMIPS.

La secuencia de ejecución para las dos instrucciones de acceso a memoria en formato I es la siguiente:

1. Leer el contenido de *rs*.
2. Sumar el número leído de *rs* con el valor inmediato en la instrucción para formar una dirección de memoria.
3. Leer de o escribir en memoria en las direcciones especificadas.
4. Para el caso de *lw*, colocar la palabra leída de la memoria en *rt*.

Observe que los primeros dos pasos de esta secuencia son idénticos a los de las instrucciones ALU en formato I, como lo es el último paso de *lw*, que involucra la escritura de un valor en *rt* (sólo se escriben los datos leídos de memoria, en vez del resultado calculado por la ALU).

El conjunto final de instrucciones en la tabla 13.1 trata con la transferencia de control condicional o incondicional a una instrucción distinta a la siguiente en secuencia. Recuerde que la bifurcación a una

dirección destino se especifica mediante un *offset* relativo al valor del contador de programa incrementado, o $(PC) + 4$. Por tanto, si la intención de la bifurcación es saltar condicionalmente a la siguiente instrucción, el valor *offset* + 1 aparecerá en el campo inmediato de la instrucción *branch*. Esto último sucede porque el *offset* se especifica en términos de palabras relativas a la dirección de memoria $(PC) + 4$. Por otra parte, para bifurcación hacia atrás, a la instrucción anterior, el valor de *offset* proporcionado en el campo inmediato de la instrucción será -2 , que resulta en la bifurcación a la dirección blanco $(PC) + 4 - 2 \times 4 = (PC) - 4$.

Para dos de las tres instrucciones *branch* (*beq*, *bne*), se comparan los contenidos de *rs* y *rt* para determinar si se satisface la condición de bifurcación. Para el caso de que la condición se sostenga, el campo inmediato se agrega a $(PC) + 4$ y el resultado se escribe de vuelta en *PC*; de otro modo, $(PC) + 4$ se escribe de vuelta en *PC*. La instrucción *branch* restante, *bltz*, es similar, excepto que la decisión de bifurcación se basa en el bit de signo del contenido de *rs* en lugar de en la comparación de los contenidos de dos registros. Para las cuatro instrucciones *jump*, *PC* se modifica incondicionalmente para permitir que la siguiente instrucción se lea desde la dirección destino del salto (*jump target address*). La dirección destino del salto proviene de la instrucción misma (*j*, *jal*), se lee desde el registro *rs* (*jr*) o es una constante conocida asociada con la ubicación de una rutina del sistema operativo (*syscall*). Observe que, aun cuando *syscall* constituye, en efecto, una instrucción *jump*, tiene un formato R.

■ 13.2 La unidad de ejecución de instrucciones

Las instrucciones MicroMIPS se pueden ejecutar en una unidad de hardware como aquella cuya estructura y componentes se muestran en la figura 13.2. Comenzando en el extremo izquierdo, el contenido del contador del programa (*PC*) se proporciona al caché de instrucción y de la ubicación especificada se lee una palabra de instrucción. En el capítulo 18 se discutirán las memorias caché. Por el momento, vea las cachés de instrucción y datos como pequeñas unidades de memoria SRAM extremadamente rápidas que pueden tener un ritmo de acción uniforme con la rapidez de los otros componentes de la figura 13.2. Como se muestra en la figura 2.10, una unidad de memoria SRAM tiene puertos de entrada de datos y direcciones y un puerto de salida de datos. El puerto de entrada de datos del caché de instrucción no se usa dentro de la ruta de datos; por tanto, no se muestra en la figura 13.2.

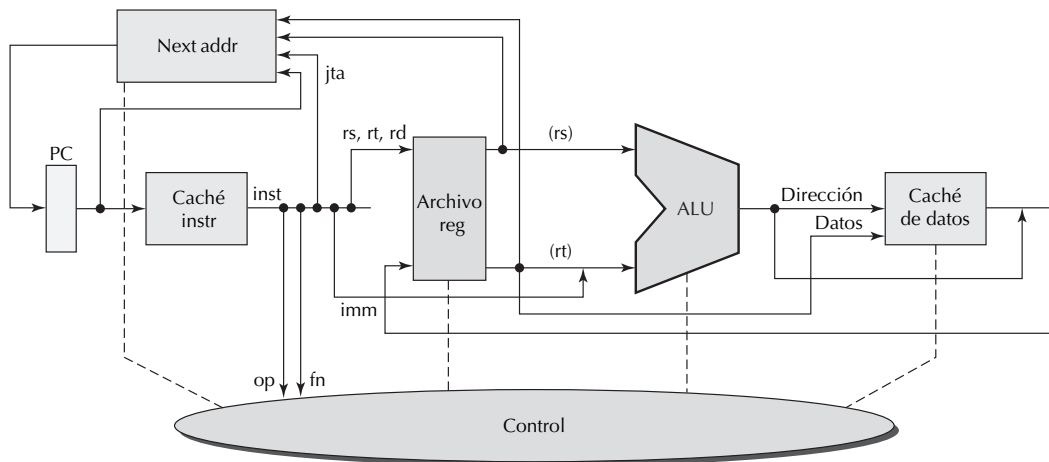


Figura 13.2 Vista abstracta de la unidad de ejecución de instrucciones para MicroMIPS. Para el nombre de los campos de instrucción, vea la figura 13.1.

Cuando una instrucción se lee del caché de instrucciones, sus campos se separan y cada uno se despacha al lugar adecuado. Por ejemplo, los campos *op* y *fn* van a la unidad de control, mientras que *rs*, *rt* y *rd* se envían al archivo de registro. La entrada superior de la ALU siempre viene del registro *rs*, mientras que su entrada inferior puede ser el contenido de *rt* o el campo inmediato de la instrucción. Para muchas instrucciones, la salida de la ALU se almacena en un registro; en tales casos, el caché de datos se puentea (*bypass*). Para el caso de las instrucciones *lw* y *sw*, se accede a la caché de datos, y el contenido de *rt* escrito en ella es para *sw* y su salida se envía al archivo de registro para *lw*.

El archivo de registro de la figura 13.2 tiene un diseño similar al de la figura 2.9b, con $h = 5$ y $k = 32$. En la figura 2.9a se muestran los detalles de la implementación del archivo de registro. En un ciclo de reloj, los contenidos de cualesquiera dos (*rs* y *rt*) de los 32 registros se pueden leer mediante los puertos de lectura, mientras que, al mismo tiempo, un tercer registro, no necesariamente distinto de *rs* o *rt*, se modifica vía el puerto de escritura. Los flip-flop que constituyen los registros se activan por flanco, de modo que la lectura desde y la escritura en el mismo registro en un solo ciclo de reloj no causa problema (figura 2.3). Recuerde que el registro \$0 representa un registro especial que siempre contiene 0 y no se puede modificar.

La ALU que se usa para la implementación MicroMIPS tiene el mismo diseño que el mostrado en la figura 10.19, excepto que no son necesarios el *corredor* y su lógica asociada, así como el bloque de detección de cero (circuito NOR de 32 entradas). La salida del *corredor* que va al mux final en la figura 10.19 se puede sustituir por la mitad superior de *y* (la entrada ALU inferior), rellena con 16 ceros a la derecha, para permitir la implementación de la instrucción *lui* (*load upper immediate*, cargar superior inmediata). Tratar *lui* como una instrucción ALU conduce a la simplificación en la unidad de ejecución porque no se necesita una ruta separada para escribir el valor inmediato extendido a la derecha en el archivo de registro. Puesto que el efecto de *lui* se puede ver como corrimiento izquierdo lógico del valor inmediato en 16 posiciones de bit, no es inadecuado usar la opción *shift* de la ALU para esta instrucción.

Para concluir la discusión preliminar de la unidad de ejecución de instrucciones, se debe especificar el flujo de datos asociado con las instrucciones *branch* y *jump*. Para las instrucciones *beq* y *bne*, se comparan los contenidos de *rs* y *rt* para determinar si la condición *branch* se satisface. Esta comparación se realiza dentro del recuadro *Next addr* (dirección siguiente) de la figura 13.2. Para el caso de *bltz*, la decisión *branch* se basa en el bit de signo del contenido de *rs* y no en la comparación de los contenidos de dos registros. De nuevo, esto último se hace dentro del recuadro *Next addr*, que también es responsable de elegir la dirección destino del salto bajo la guía de la unidad de control. Recuerde que la dirección destino del salto proviene de la instrucción misma (*j*, *jal*), y se lee del registro (*jr*) o es una constante conocida asociada con la ubicación de una rutina del sistema operativo (*syscall*).

Observe que en la figura 13.2 faltan algunos detalles. Por ejemplo, no se muestra cómo se notifica al archivo de registro en cuál registro, para el caso de que exista alguno, se escribirá (*rd* o *rt*). Estos detalles se proporcionarán en la sección 13.3.

■ 13.3 Una ruta de datos de ciclo sencillo

Ahora comienza el proceso de refinar la unidad de ejecución abstracta de la figura 13.2 hasta que se convierta en un circuito lógico concreto capaz de ejecutar las 22 instrucciones MicroMIPS. En esta sección se ignoran los bloques *Next addr* y “Control” de la figura 13.2 y se enfoca la atención en la parte media compuesta del contador de programa, caché de instrucciones, archivo de registro, ALU y caché de datos. Esta parte se conoce como *ruta de datos*. La lógica de dirección siguiente se cubrirá en la sección 13.4 y el circuito de control en la sección 13.5.

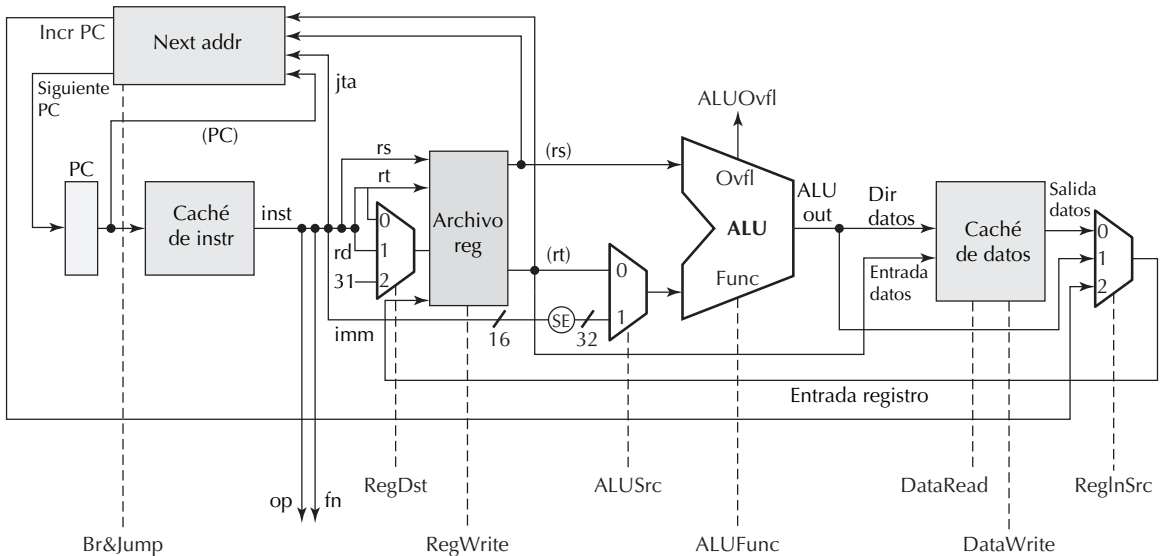


Figura 13.3 Elementos clave de la ruta de datos MicroMIPS de ciclo sencillo.

La figura 13.3 muestra algunos detalles de la ruta de datos que se pierde en la versión abstracta de la figura 13.2. Todo lo que sigue en esta sección se amolda a la parte de la ruta de datos de la figura 13.3, por la que los pasos de ejecución de instrucción proceden de izquierda a derecha. Ya se describieron los cuatro bloques principales que aparecen en esta ruta de datos. Ahora se explicará la función de los tres multiplexores que se usan en la entrada al archivo de registro, en la entrada inferior de la ALU, así como en las salidas de la ALU y en el caché de datos. Entender la importancia de estos multiplexores constituye una clave para comprender además de la ruta de datos de la figura 13.3 cualquier ruta de datos en general.

El multiplexor en la entrada al archivo de registro permite que *rt*, *rd* o *\$31* se usen como el índice del registro de destino en el cual se escribirá un resultado. Un par de señales lógicas *RegDst*, proporcionadas por la unidad de control, dirigen la selección. *RegDst* se fija a 00 para seleccionar *rt*, 01 para *rd* y 10 para *\$31*; esta última opción es necesaria para ejecutar *jal*. Desde luego, no toda instrucción escribe un valor en un registro. Escribir en un registro requiere que la unidad de control postule la señal de control *RegWrite*; de otro modo, sin importar el estado de *RegDst*, nada se escribe en el archivo de registro. Observe que los registros, *rs* y *rt*, se leen para cada instrucción, aun cuando puedan no ser necesarios en todos los casos. Por tanto, no se muestra señal de control de lectura para el archivo de registro. En este contexto, la caché de instrucciones no recibe ninguna señal de control, porque en cada ciclo se lee una instrucción (es decir, la señal de reloj sirve como control de lectura para la caché de instrucciones).

El multiplexor en la entrada inferior a la ALU permite que la unidad de control elija el contenido de *rt* o la versión extendida en signo de 32 bits del operando inmediato de 16 bits como la segunda entrada de ALU (la primera entrada, o entrada superior, siempre proviene de *rs*). Lo anterior se controla mediante la postulación o no postulación de la señal de control *ALUSrc*. Para el caso de que esta señal no se postule (tiene un valor 0), se usará el contenido de *rt* como la entrada inferior a la ALU; de otro modo, se usa el operando inmediato, extendido en signo a 32 bits. La extensión de signo del operando inmediato se realiza mediante el bloque circular etiquetado “SE” en la figura 13.3. (Vea el problema 13.17.)

Finalmente, el multiplexor de la extrema derecha en la figura 13.3 permite que la palabra proporcionada por la caché de datos, la salida de la ALU o el valor PC incrementado se envíen al archivo de registro para escritura (la última opción se necesita para `jal`). La elección se efectúa mediante un par de señales de control, `RegInSrc`, que se fijan a 00 para elegir la salida de caché de datos, 01 para la salida ALU y 10 para el valor PC incrementado proveniente del bloque *next-address* (dirección siguiente).

La ruta de datos de la figura 13.3 es capaz de ejecutar una instrucción por ciclo de reloj; de ahí la denominación “ruta de datos de ciclo sencillo”. Con cada tic del reloj, se carga una nueva dirección en el contador de programa, ello propicia que aparezca una nueva instrucción en la salida de la caché de instrucciones después de un corto retardo de acceso. Los contenidos de los diversos campos en la instrucción se envían a los bloques relevantes, incluida la unidad de control, que decide (con base en los campos `op` y `fn`) qué operación realizar por cada bloque.

Conforme los datos provenientes de `rs` y `rt`, o `rs` e `imm` extendida en signo, pasan a través de la ALU, se realiza la operación especificada por las señales de control `ALUFunc` y el resultado aparece en la salida ALU. Para el caso de que se ignoren las señales de control relacionadas con *shift* (`Const'Var`, la función *Shift*) de la figura 10.19 que no se necesitan para MicroMIPS, el haz de señal de control `ALUFunc` contiene cinco bits: un bit para control de sumador (`Add'Sub`), dos bits para controlar la unidad lógica (función `Logic`) y dos bits para controlar el multiplexor de la extrema derecha en la figura 10.19 (clase `Función`). En la figura 13.3 se muestra la señal de salida ALU que indica desbordamiento en suma o resta, aunque no se usa en este capítulo.

Para el caso de las instrucciones aritméticas y lógicas, el resultado de ALU debe almacenarse en el registro de destino y, por tanto, se adelanta al archivo de registro mediante la ruta de retroalimentación cerca del fondo de la figura 13.3. Respecto de las instrucciones de acceso a memoria, la salida ALU representa una dirección de datos para escribir en la caché de datos (`DataWrite` postulada) o lectura desde ella (`DataRead` postulada). En el último caso, la salida de caché de datos, que aparece después de una latencia corta, se envía mediante la ruta de retroalimentación inferior hacia el archivo de registro para escritura. Finalmente, cuando la instrucción ejecutada es `jal`, el valor del contador de programa incrementado, $(PC) + 4$, se almacena en el registro `$31`.

13.4 Bifurcación y saltos

Esta sección se dedica al diseño del bloque *next-address* que aparece en la parte superior izquierda de la figura 13.3.

La dirección siguiente a cargar en el contador del programa se deriva en una de cinco formas, según la instrucción que se ejecutará y los contenidos de los registros en los que se basa la condición de bifurcación. En virtud de que las instrucciones son palabras almacenadas en ubicaciones de memoria con direcciones que son múltiplos de 4, los dos LSB del contador de programa siempre se fijan a 0. Por tanto, en la discusión siguiente, $(PC)_{31:2}$ se refiere a los 30 bits superiores del contador de programa, que es la parte del PC que se modifica por la lógica de *next-address*. Con esta convención, sumar 4 al contenido del contador de programa se realiza al calcular $(PC)_{31:2} + 1$. Además, el valor inmediato, que por definición se debe multiplicar por 4 antes de sumarlo al valor incrementado del contador de programa, se suma a esta parte superior sin cambio. Las cinco opciones para el contenido de $(PC)_{31:2}$ son:

$(PC)_{31:2} + 1$	Opción por defecto
$(PC)_{31:2} + 1 + imm$	Cuando la instrucción es una bifurcación y la condición se satisface
$(PC)_{31:28} jta$	Cuando la instrucción es <code>j</code> o <code>jal</code>
$(rs)_{31:2}$	Cuando la instrucción es <code>jr</code>
<code>SysCallAddr</code>	Dirección de arranque de una rutina de sistema operativo

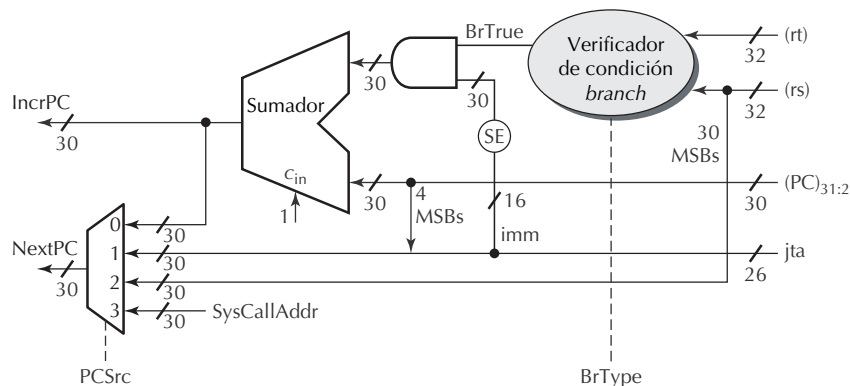


Figura 13.4 Lógica de dirección siguiente para MicroMIPS (vea la parte superior de la figura 13.3).

Las primeras dos opciones se combinan con el uso de un sumador con su entrada inferior unida a $(PC)_{31:2}$, su entrada superior conectada a imm (extendido en signo a 30 bits) o 0, dependiendo de si se realizará o no una bifurcación, así como su señal de entrada de acarreo postulada permanentemente (figura 13.4). Por tanto, este *sumador de dirección* calcula $(PC)_{31:2} + 1$ o $(PC)_{31:2} + 1 + imm$, que se escriben en PC a menos que se ejecute una instrucción *jump*. Observe que cuando la instrucción no es una bifurcación, la salida de este sumador es $(PC)_{31:2} + 1$. Por tanto, esta salida, con dos 0 unidos a su derecha se puede usar como el valor incrementado del contador de programa que se debe almacenar en §31, como parte de la ejecución de la instrucción *jal*.

Con referencia en la figura 13.4, el verificador de la condición de bifurcación comprueba el predicado de branch, que es $(rs) = (rt)$, $(rs) \neq (rt)$, o $(rs) < 0$, y postula su salida *BrTrue* si una de estas condiciones se satisface y se ejecuta la correspondiente instrucción *branch*. La última información la proporcionan el par de señales *BrType* de la unidad de control. Finalmente, el par de señales *PCSrc*, también proporcionadas por la unidad de control, dirigen al multiplexor del extremo izquierdo de la figura 13.4 para que envíe una de sus cuatro entradas a escribirse en los 30 bits superiores del PC. Estas señales se fijan en 00 la mayor parte del tiempo (para todas las instrucciones distintas a las cuatro *jump*); se fijan a 01 tanto para *j* como para *jal*, 10 para *jr* y 11 para *syscall*.

13.5 Derivación de las señales de control

La ejecución adecuada de las instrucciones MicroMIPS requiere que el circuito de control asigne valores apropiados a las señales de control que se muestran en la figura 13.3 y en la parte que se elabora en la figura 13.4. El valor para cada señal es una función de la instrucción que se ejecutará; por tanto, se determina de manera única mediante sus campos *op* y *fn*. La tabla 13.2 contiene una lista de todas las señales de control y sus definiciones. En la columna de la extrema izquierda de la tabla 13.2, se nombra el bloque de la figura 13.3 al que se relacionan las señales. La columna 2 menciona las señales conforme aparecen en la figura 13.3 y, si es aplicable, asigna nombres a los componentes de las señales. Por ejemplo, *ALUFunc* tiene tres componentes (figura 10.19): *AddSub*, *LogicFn* y *FnClass*. Las dos últimas son señales de dos bits, con sus bits indexados 1 (MSB) y 0 (LSB). Las otras columnas de la tabla 13.2 especifican el significado asociado con cada valor de señal de control o establecimiento. Por ejemplo, la siguiente a la última línea en la tabla asigna patrones de bit distintos o códigos a los tres tipos diferentes de *branch* (*beq*, *bne*, *bltz*) y a todos los otros casos en que no ocurrirá bifurcación.

■ **TABLA 13.2** Señales de control para la implementación MicroMIPS de ciclo sencillo.

Bloque	Señal de control		0	1	2	3
Reg file	RegWrite		No escribe	Escribe		
	RegDst ₁ , RegDst ₀		rt	rd	\$31	
	RegInSrc ₁ , RegInSrc ₀		Data out	ALU out	IncrPC	
	ALUSrc		(rt)	imm		
ALU	ALUFunc	Add/Sub	Suma	Resta		
		LogicFn ₁ , LogicFn ₀	AND	OR	XOR	NOR
		FnClass ₁ , FnClass ₀	lui	Fija menor	Aritmética	Lógica
Data cache	DataRead		No lee	Lee		
	DataWrite		No escribe	Escribe		
Next addr	Br&Jump	BrType ₁ , BrType ₀ PCSrc1, PCSrc0	No bifurca IncrPC	beq jta	bne (rs)	bltz SysCallAddr

■ **TABLA 13.3** Establecimientos de señales de control para la unidad de ejecución de instrucciones MicroMIPS de ciclo sencillo.*

Instrucción	op	fn	RegWrite	RegDst	RegInSrc	ALUSrc	Add/Sub	LogicFn	FnClass	DataRead	DataWrite	BrType	PCSrc
Carga superior inmediata	001111		1	00	01	1			00	0	0	00	00
Suma	000000	100000	1	01	01	0	0		10	0	0	00	00
Resta	000000	100010	1	01	01	0	1		10	0	0	00	00
Fija menor que	000000	101010	1	01	01	0	1		01	0	0	00	00
Suma inmediato	001000		1	00	01	1	0		10	0	0	00	00
Fija menor que inmediato	001010		1	00	01	1	1		01	0	0	00	00
AND	000000	100100	1	01	01	0		00	11	0	0	00	00
OR	000000	100101	1	01	01	0		01	11	0	0	00	00
XOR	000000	100110	1	01	01	0		10	11	0	0	00	00
NOR	000000	100111	1	01	01	0		11	11	0	0	00	00
AND inmediata	001100		1	00	01	1		00	11	0	0	00	00
OR inmediata	001101		1	00	01	1		01	11	0	0	00	00
XOR inmediata	001110		1	00	01	1		10	11	0	0	00	00
Carga palabra	100011		1	00	00	1	0		10	1	0	00	00
Almacena palabra	101011		0			1	0		10	0	1	00	00
Salto	000010		0							0	0		01
Salta registro	000000	001000	0							0	0		10
Branch en menor que 0	000001		0							0	0	11	00
Branch en igual	000100		0							0	0	01	00
Branch en no igual	000101		0							0	0	10	00
Saltar y ligar	000011		1	10	10					0	0	00	01
Petición de sistema	000000	001100	0							0	0		11

*Nota: Las entradas en blanco son del tipo “no importa”.

Con base en las definiciones de la tabla 13.2 y la comprensión de lo que se necesita hacer para ejecutar cada instrucción, se construye la tabla 13.3, en la que se especifican los valores de las 17 señales de control para cada una de las 22 instrucciones MicroMIPS.

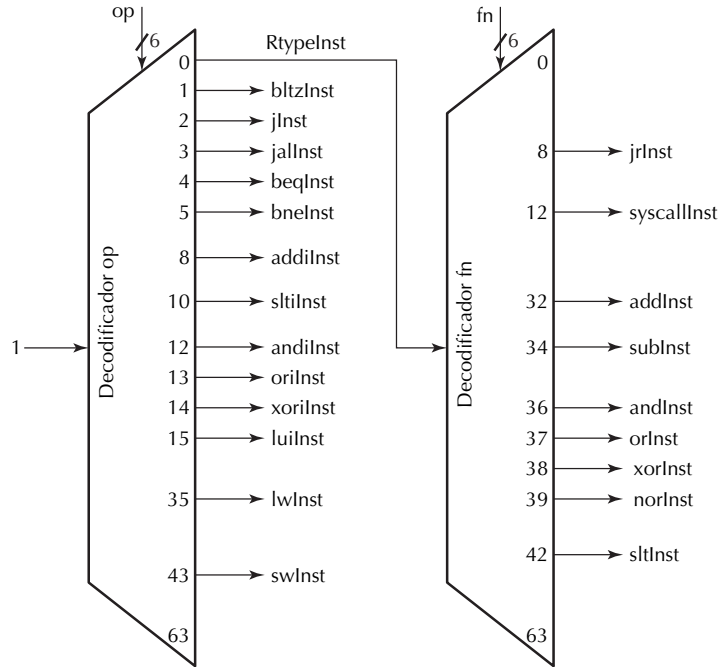


Figura 13.5 Decodificador de instrucción para MicroMIPS construido con dos decodificadores de 6 a 64.

En la tabla 13.2 se define cada una de las 17 señales de control como función lógica de 12 bits de entrada (op y fn). Es fácil derivar expresiones lógicas para estas señales en términos de los 12 bits de entrada $op_5, op_4, \dots, op_0, fn_5, fn_4, \dots, fn_0$. El inconveniente de tales enfoques *ad hoc* es que, si más tarde se decide modificar el conjunto de instrucciones de la máquina o agregarle nuevas instrucciones, todo el diseño se debe modificar.

En este contexto, con frecuencia se prefiere un enfoque de dos pasos para la síntesis de los circuitos de control. En el primero, el conjunto de instrucciones se decodifica y se postula una señal lógica diferente para cada instrucción. La figura 13.5 muestra un decodificador para el conjunto de instrucciones de MicroMIPS. El campo op de seis bits se proporciona a un decodificador de 6 a 64 que postula una de sus salidas dependiendo del valor de op . Por ejemplo, la salida 1 del decodificador op corresponde a la instrucción *bltz*; por tanto, se le da el nombre simbólico “*bltzInst*”. Esto último hace fácil recordar cuándo se postulará cada señal. La salida 0 del decodificador op va hacia un segundo decodificador. De nuevo, las salidas de este decodificador fn se etiquetan con los nombres que corresponden a las instrucciones que representan.

Ahora, cada una de las 17 señales de control requeridas se pueden formar como la OR lógica de un subconjunto de las salidas de decodificador de la figura 13.5. Como consecuencia de que algunas de las señales de la tabla 13.3 tienen muchas entradas 1 en sus tablas de verdad, lo que de cualquier forma requiere operación OR multinivel, es conveniente definir muchas señales auxiliares que luego se usen en la formación de las señales de control principales. Defina tres señales auxiliares del modo siguiente:

$$\text{arithInst} = \text{addInst} \vee \text{subInst} \vee \text{sltInst} \vee \text{addiInst} \vee \text{sltiInst}$$

$$\text{logicInst} = \text{andInst} \vee \text{orInst} \vee \text{xorInst} \vee \text{norInst} \vee \text{andiInst} \vee \text{oriInst} \vee \text{xoriInst}$$

$$\text{immInst} = \text{luiInst} \vee \text{addiInst} \vee \text{sltiInst} \vee \text{andiInst} \vee \text{oriInst} \vee \text{xoriInst}$$

Entonces, por ejemplo:

RegWrite = luiInst ∨ arithInst ∨ logicInst ∨ lwInst ∨ jalInst

ALUSrc = immInst ∨ lwInst ∨ swInst

Add/Sub = subInst ∨ sltInst ∨ sltiInst

DataRead = lwInst

PCSrc₀ = jInst ∨ jalInst ∨ syscallInst

Se deja como ejercicio la derivación de las expresiones lógicas para las señales de control restantes de las figuras 13.3 y 13.4.

■13.6 Rendimiento del diseño de ciclo sencillo

La implementación MicroMIPS de ciclo sencillo que se discutió en las anteriores secciones también se puede referir como una implementación de estado sencillo o sin estado, en la que el circuito de control es meramente combinacional y no requiere recordar nada de un ciclo de reloj al siguiente. Toda la información que se necesita para una operación adecuada en el siguiente ciclo de reloj se lleva en el contador de programa. Dado de que una nueva instrucción se ejecuta en cada ciclo de reloj, se tiene CPI = 1. Todas las instrucciones comparten los pasos *fetch* y de acceso al registro. La decodificación de instrucciones se traslapa completamente con el acceso al registro, de modo que no implica latencia adicional.

El ciclo de reloj está determinado por el tiempo de ejecución más largo para una instrucción, que a su vez depende de la latencia de propagación de señal a través de la ruta de datos de la figura 13.3. Con el conjunto de instrucciones simples, la instrucción *lw*, que necesita una operación ALU, acceso a caché de datos y escritura en registro, es probable que sea la más lenta de las instrucciones. Ninguna otra instrucción necesita los tres pasos. Suponga las siguientes latencias de peor caso para los bloques en la ruta de datos, cuya suma produce la latencia de ejecución para la instrucción *lw*:

Acceso de instrucción	2 ns
Leer registro	1 ns
Operación ALU	2 ns
Acceso a caché de datos	2ns
Escritura de registro	<u>1ns</u>
Total	8 ns

Lo anterior corresponde a una velocidad de reloj de 125 MHz y rendimiento de 125 MIPS. Si el tiempo de ciclo se pudiera hacer variable para ajustar al tiempo real necesario para cada instrucción, se obtendría un tiempo de ejecución de instrucción promedio un poco mejor. La figura 13.6 muestra las rutas de propagación de señal crítica para las diversas clases de instrucción en MicroMIPS. Aquí se supondrá que todas las instrucciones *jump* representan la variedad más rápida que no necesita acceso al registro (*j* o *syscall*). Esto conduce a una estimación de mejor caso para el tiempo de ejecución de instrucción promedio, pero el verdadero promedio no será muy diferente, de acuerdo con el pequeño porcentaje de instrucciones *jump*. Con la siguiente mezcla de instrucciones típica se puede derivar un tiempo de ejecución de instrucción promedio.

Tipo R	44%	6 ns	No acceso a caché de datos
Load	24%	8 ns	Ver análisis precedente
Store	12%	7 ns	No escritura en registro
Branch	18%	5 ns	<i>Fetch</i> + lectura registro + formación siguiente dirección
Jump	2%	<u>3 ns</u>	<i>Fetch</i> + decodificación instrucción
Promedio ponderado \cong 6.36 ns			

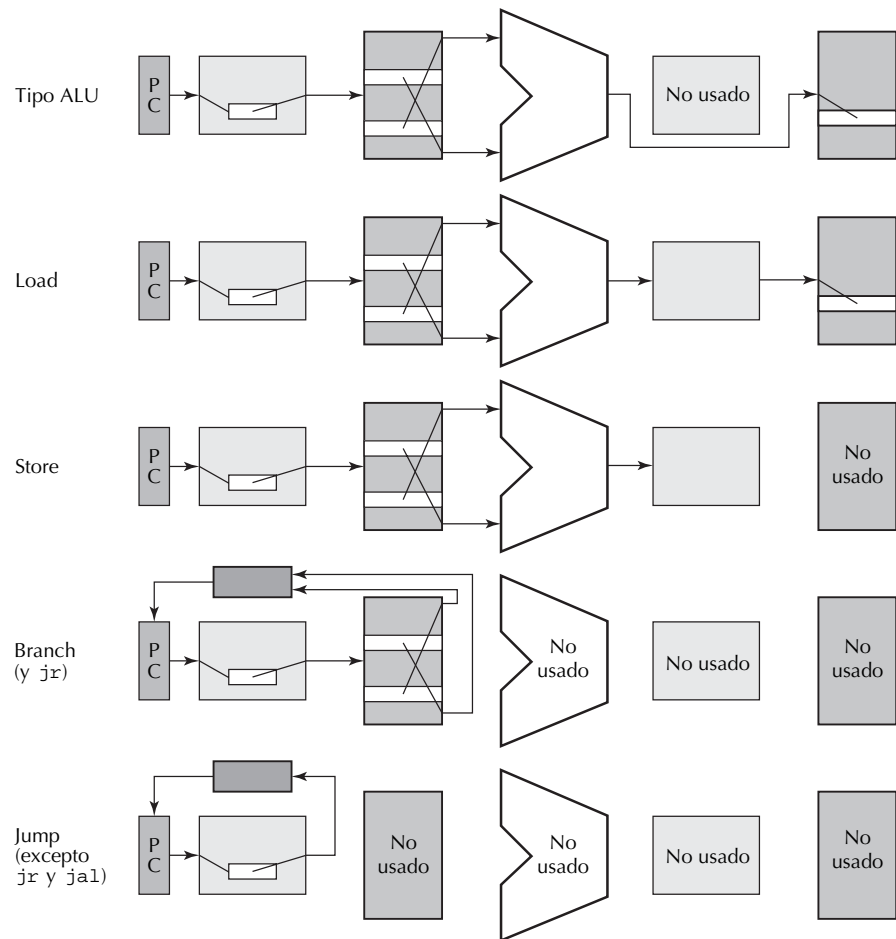


Figura 13.6 Despliegue de la ruta de datos MicroMIPS (mediante el paso de escritura de registro como un bloque separado) para permitir mejor visualización de las latencias de ruta crítica para diversas clases de instrucción.

De este modo, la implementación de un ciclo sencillo con un tiempo de ciclo fijo es más lento que la implementación ideal con tiempo de ciclo variable que podría lograr un rendimiento promedio de 157 MIPS. Sin embargo, lo último no es práctico y aquí sólo se discute para mostrar que no se puede obtener una implementación de hardware MicroMIPS con mayor rendimiento basado en la filosofía de control de ciclo sencillo. A continuación se discuten las desventajas del diseño de ciclo sencillo.

Como ya se observó, el tiempo de ciclo de reloj de una máquina con control de ciclo sencillo se determina mediante la instrucción más compleja o más lenta. Por tanto, con tal implementación se aumentarían los retardos de operaciones simples para acomodar operaciones complejas en un ciclo de reloj. Para el caso de MicroMIPS, fue necesario fijar el periodo de reloj a 8 ns, que corresponde al tiempo de ejecución de la instrucción MicroMIPS más lenta. Esto último provocó que las instrucciones más rápidas, que teóricamente podrían ejecutarse en 3, 5, 6 o 7 ns, también tomarán 8 ns.

De hecho, se es más bien afortunado de que, excepto por las instrucciones *jump* que representan una fracción mínima de las encontradas, los tiempos de ejecución para las instrucciones MicroMIPS

no se desvían significativamente del promedio, ello hace el control de ciclo sencillo con duración de ciclo fija bastante competitiva con una implementación ideal de tiempo de ciclo variable. Si la mezcla de instrucciones ofrecida en esta sección es típica de las aplicaciones a correr en la máquina, más de un tercio de las instrucciones MicroMIPS de ciclo sencillo se ejecutan cerca de sus latencias mínimas cuando se usa un tiempo de ciclo de reloj de 8 ns, mientras que 80% sufre de no más de 33% de frenado como resultado de la implementación de ciclo sencillo.

Si MicroMIPS hubiese incluido instrucciones más complejas, como multiplicación, división o aritmética de punto flotante, la desventaja del control de ciclo sencillo habría sido más pronunciada. Por ejemplo, si la latencia de una instrucción más compleja como la división fuese cuatro veces la de la suma, la implementación de control de ciclo sencillo implicaría que la misma operación común de suma se debe frenar por un factor de al menos 4, ello afecta el rendimiento. Como se sabe, a partir de la discusión en la sección 8.5, lo anterior está en conflicto directo con las instrucciones multiplicación y división a una unidad distinta a la ALU principal (figura 5.1). La unidad multiplicar/dividir puede realizar estas operaciones más complejas, mientras la ALU continúa leyendo y ejecutando instrucciones más simples. En tanto se permita un tiempo adecuado antes de copiar los resultados de multiplicación o división de los registros H_i y L_o en el archivo de registro general, no surgirá problema entre estas dos unidades aritméticas independientes.

La analogía siguiente ayuda a seguir la dirección hacia la implementación del control multiciclo en el capítulo 14. Considere la agenda de un dentista. En lugar de ordenar todas las citas a la hora, porque algunos pacientes requieren una hora completa de trabajo, éstas se pueden asignar en incrementos estándar de 15 minutos. Un paciente que llega para una revisión de rutina podría usar sólo 15 minutos del tiempo del dentista. En este sentido, las personas que requieren más atención o procedimientos complejos pueden recibir incremento de tiempo múltiples. De esta forma, el dentista puede atender a más pacientes y tendrá menos tiempo desperdiciado entre pacientes.

PROBLEMAS

13.1 Detalles de ruta de datos

- Etiquete cada una de las líneas de la figura 13.2 con el número de señales binarias que representa.
- Repita la parte a) para la figura 13.3.

13.2 Opciones de selección en ruta de datos

Sean M_1 , M_2 y M_3 , de izquierda a derecha, los tres multiplexores de la figura 13.3.

- M_1 tiene tres arreglos, mientras que M_2 tiene dos; por tanto, existen seis combinaciones cuando se toman en conjunto los arreglos M_1 y M_2 . Para cada una de esas combinaciones, indique si alguna vez se usa y, si fuera así, para ejecutar cuál(es) instrucción(es) MicroMIPS.
- Repita la parte a) para M_2 y M_3 (seis combinaciones).
- Repita la parte a) para M_1 y M_3 (nueve combinaciones).

13.3 Verificador de condición *branch*

Con base en la descripción de la sección 13.4 y las codificaciones de señal definidas en la tabla 13.2, presente un diseño lógico completo para el verificador de condición *branch* de la figura 13.4.

13.4 Lógica de dirección siguiente

Considere el siguiente diseño alternativo para la lógica de dirección siguiente de MicroMIPS. El arreglo de 30 compuertas AND que se muestra cerca de lo alto de la figura 13.4 se elimina y el valor inmediato, extendido en signo a 30 bits, se conecta directamente a la entrada superior del sumador. Se introduce un incrementador separado para calcular $(PC)_{31:2} + 1$. Entonces se usa la señal BrTrue para controlar un multiplexor que permita que la salida IncrPC se tome del sumador o el incrementador recientemente introducido. Compare este diseño alternativo con el diseño original de la figura 13.4 en términos de ventajas y desventajas potenciales.

13.5 Formato de instrucción MicroMIPS

Suponga que se le proporciona una pizarra en blanco para rediseñar el formato de instrucción para MicroMIPS. El conjunto de instrucciones se define en la tabla 13.1. El único requisito es que cada instrucción contenga el número adecuado de campos de especificación de registro de cinco bits y un campo inmediato/*offset* de 16 bits donde se necesite. En particular, el ancho del campo *jta* puede cambiar. ¿Cuál es el número mínimo de bits que se requiere para codificar todas las instrucciones MicroMIPS en un formato de ancho fijo? *Sugerencia:* En virtud de que existen siete instrucciones diferentes que necesitan dos campos registro y un campo inmediato, puede establecer una cota inferior en el ancho.

13.6 Valores de señal de control

- Al examinar la tabla 13.3, se observa que para cualquier par de hileras, existe al menos una columna en la que los establecimientos de señal de control son diferentes (uno se fija a 0, el otro a 1). ¿Por qué esto último no causa sorpresa?
- Identifique tres pares de hileras, donde cada par difiera exactamente en un valor de bit de control. Explique cómo esta diferencia en los valores del bit de control provoca distintos comportamientos de ejecución.

13.7 Derivación de señales de control

Las expresiones lógicas para cinco de las 17 señales de control citadas en las tablas 13.2 y 13.3 se proporcionan al final de la sección 13.5. Proporcione expresiones lógicas que definan las otras 12 señales de control.

13.8 Rendimiento del diseño de ciclo sencillo

Discuta los efectos de los siguientes cambios en los resultados de rendimiento obtenidos en la sección 13.6:

- Mediante la reducción del tiempo de acceso al registro de 1 ns a 0.5 ns.
- Con el mejoramiento de la latencia ALU de 2 ns a 1.5 ns.
- Con el uso de memorias caché con tiempo de acceso de 3 ns en lugar de 2 ns.

13.9 Manejo de excepción

- Suponga que el desbordamiento ALU causará que se invoque una rutina especial del sistema operativo en

la ubicación *ExceptionHandler* de la memoria. Discuta cambios en la unidad de ejecución de instrucción MicroMIPS para acomodar este cambio.

- Discuta cómo se puede hacer una provisión similar para desbordamiento en cálculo de dirección (observe que las direcciones son números sin signo).

13.10 Decodificación de instrucción

Suponga que el decodificador de instrucción de la implementación MicroMIPS de ciclo sencillo se diseñara directamente de la tabla 13.3, en lugar de que tenga como base el proceso de dos etapas de decodificación completa seguido por la operación OR (figura 13.5). Escriba la expresión lógica más simple posible para cada una de las 17 señales de control mencionadas en la tabla 13.3 (partes a–q del problema, en orden de izquierda a derecha). Las entradas en blanco y perdidas en la tabla se pueden considerar como del tipo “no importa”.

13.11 Decodificación de instrucción

- Muestre cómo se puede sustituir, usando sólo una compuerta AND adicional de tres entradas, el decodificador de operación del lado izquierdo de la figura 13.5, por un decodificador mucho más pequeño 4 a 16. *Sugerencia:* Reste 32 de 35 y 43.
- Muestre cómo se puede sustituir el decodificador de función, en el lado derecho de la figura 13.5, por un decodificador 4 a 16.

13.12 Señales de control para instrucciones *shift*

Extienda la tabla 13.3 con líneas correspondientes a las siguientes instrucciones nuevas de la tabla 6.2 que se pueden agregar a la implementación MicroMIPS de ciclo sencillo. Justifique sus respuestas.

- Corrimiento izquierdo lógico (*sll*).
- Corrimiento derecho lógico (*srl*).
- Corrimiento derecho aritmético (*sra*).
- Corrimiento izquierdo lógico variable (*sllv*).
- Corrimiento derecho lógico variable (*srlv*).
- Corrimiento derecho aritmético variable (*srav*).

13.13 Manejo de otras instrucciones

Explique los cambios necesarios en la ruta de datos de ciclo sencillo y los establecimientos de señal de control

asociados para agregar las siguientes nuevas instrucciones de la tabla 6.2 a la implementación MicroMIPS. Justifique sus respuestas.

- a) Cargar byte (1b).
- b) Cargar byte sin signo (1bu).
- c) Corrimiento derecho aritmético (sra).

13.14 Manejo de instrucciones multiplicar/dividir

Suponga que quiere aumentar MicroMIPS con una unidad multiplicar/dividir para permitirle ejecutar las instrucciones multiplicación y división en la tabla 6.2 junto con las instrucciones asociadas `mfhi` y `mflo`. Además de `Hi` y `Lo`, la unidad multiplicar/dividir tiene un registro `Md`, que retiene el multiplicando/divisor durante la ejecución de la instrucción. El multiplicador/dividendo se almacena, y el cociente se desarrolla, en `Lo`. A la unidad multiplicar/dividir se le debe proporcionar sus operandos y algunos bits de control que indiquen cuál operación se realiza. Entonces la unidad opera de manera independiente a partir de la principal ruta de datos durante varios ciclos de reloj, eventualmente obtiene los resultados requeridos. No se preocupe por la operación de la unidad más allá de la configuración inicial. Proponga cambios en la unidad de ejecución de ciclo sencillo para acomodar esos cambios.

13.15 Adición de otras instrucciones

Se pueden añadir otras instrucciones al conjunto de instrucciones de MicroMIPS. Considere las pseudoinstrucciones de la tabla 7.1 y suponga que se les quiere incluir

en MicroMIPS como instrucciones regulares. En cada caso, elija una codificación apropiada para la instrucción y especifique todas las modificaciones requeridas en la ruta de datos de ciclo sencillo y circuitos de control asociados. Asegúrese de que las codificaciones elegidas no están en conflicto con las otras instrucciones MicroMIPS mencionadas en las tablas 6.2 y 12.1.

- a) Mover (move).
- b) Cargar inmediato (li).
- c) Valor absoluto (abs).
- d) Negar (neg).
- e) Not (not).
- f) Bifurcación menor que (blt).

13.16 URISC

Considere el procesador URISC descrito en la sección 8.6 [Mava88]. ¿Cuántos ciclos de reloj se necesitan para que URISC ejecute una instrucción, si supone que a la memoria se puede acceder en un ciclo de reloj?

13.17 ALU para MicroMIPS

La descripción de la ALU MicroMIPS al final de la sección 13.2 omite un detalle: que la unidad lógica debe experimentar el efecto de extensión de signo en un operando inmediato.

- a) Presente el diseño de la unidad lógica dentro de la ALU con esta provisión.
- b) Especule acerca de posibles razones para esta elección de diseño (es decir, extensión de signo en todos los casos, seguido de posible deshacer).

REFERENCIAS Y LECTURAS SUGERIDAS

- [Mava88] Mavaddat, F. y B. Parhami, "URISC: The Ultimate Reduced Instruction Set Computer", *Int. J. Electrical Engineering Education*, vol. 25, pp. 327-334, 1988.
- [MIPS] Tecnologías MIPS, sitio web. Siga las ligas de arquitectura y documentación en: <http://www.mips.com/>
- [Patt98] Patterson, D. A. y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2a. ed., 1998.
- [Wake01] Wakerly, J. F., *Digital Design: Principles and Practices*, 3a. ed. actualizada, Prentice Hall, 2001.

SÍNTESIS DE UNIDAD DE CONTROL

“No tener control sobre los sentidos es como navegar en un barco sin timón, que se encuentra condenado a romperse en pedazos cuando se estrelle contra la primera roca que halle.”

Mahatma Gandhi

“Microprogramación representa la implementación de sistemas esperanzadoramente razonables mediante la interpretación de máquinas irracionales.”

R. F. Rosin

TEMAS DEL CAPÍTULO

- 14.1** Una implementación multiciclo
- 14.2** Ciclo de reloj y señales de control
- 14.3** La máquina de control de estado
- 14.4** Rendimiento del diseño multiciclo
- 14.5** Microprogramación
- 14.6** Excepciones

El circuito de control sin memoria para la implementación de ciclo sencillo del capítulo 13 forma todas las señales de control como funciones de ciertos bits dentro de la instrucción. Lo anterior es bueno para un conjunto limitado de instrucciones, la mayoría de las cuales se ejecutan en casi el mismo tiempo. Cuando las instrucciones son más variadas en complejidad o cuando algunos recursos se deben usar más de una vez durante la misma instrucción, se solicita una implementación multiciclo. El circuito de control de tal implementación multiciclo es una máquina de estado, con un número de estados (*states*) para ejecución normal y estados adicionales para manejo de excepciones. En este capítulo se deriva una implementación multiciclo de control para MicroMIPS y se muestra que la ejecución de cada instrucción ahora se vuelve un “programa hardware” (o microprograma) que, como programa normal, tiene ejecución secuencial, bifurcación y, acaso, incluso *ciclos* y llamadas de procedimiento.

■ 14.1 Implementación multiciclo

Como aprendió en el capítulo 13, la ejecución de cada instrucción MicroMIPS abarca un conjunto de acciones, como acceso a memoria, lectura de registro y operación ALU. De acuerdo con las suposiciones de la sección 13.5, cada una de estas acciones tarda 1-2 ns en completarse. La operación de ciclo

sencillo requiere que la suma de estas latencias se tome como el periodo de reloj. Con diseño multiciclo, en un ciclo de reloj se realiza un subconjunto de acciones requeridas por una instrucción. En consecuencia, el ciclo de reloj se puede hacer mucho más corto, con muchos ciclos necesarios ejecutando una sola instrucción. Esto último es análogo a un consultorio dental que asigna tiempo a los pacientes en múltiplos de 15 minutos, dependiendo de la cantidad de trabajo que esté anticipado. Para permitir la operación multiciclo, los valores intermedios de un ciclo deben mantenerse en registros de modo que estén disponibles para examen en cualquier ciclo de reloj subsecuente donde se necesite.

Se puede usar una implementación multiciclo por razones de mayor rapidez o economía. La operación más rápida resulta de tomar un periodo de reloj más corto y usar un número variable de ciclos de reloj por instrucción; por tanto, cada instrucción tanto tiempo como necesite para los diversos pasos de ejecución en lugar de cierto tiempo como la instrucción más lenta (figura 14.1). El costo de implementación más corto resulta de ser capaz de usar algunos recursos más de una vez en el curso de la ejecución de la instrucción; por ejemplo, el mismo sumador que se usa para ejecutar la instrucción *add* se puede usar para calcular la *branch target address* (dirección destino de bifurcación) o para incrementar el contador de programa.

En la figura 14.2 se muestra una vista abstracta de una ruta de datos multiciclo. Son notables muchas características de esta ruta. Primero, los dos bloques de memoria (caché de instrucción y caché de datos) de la figura 13.2 se han fundido en un solo bloque de caché. Cuando una palabra se lee del caché, se debe conservar en un registro para su uso en ciclos subsecuentes. La razón para tener dos registros (de instrucción y de datos) entre la caché y el archivo de registro es que, cuando la instrucción se lee, se debe mantener para todos los ciclos restantes en su ejecución con el propósito de generar adecuadamente las señales de control. De este modo, se necesita un segundo registro para la lectura de datos asociados con *lw*. Otros tres registros (*x*, *y* y *z*) también sirven al propósito de retener información entre ciclos. Observe que, excepto para PC y el registro de instrucción, todos los registros se cargan en cada ciclo de reloj; de ahí la ausencia de control explícito para la carga de estos registros. Como consecuencia de que el contenido de cada uno de estos registros se necesita sólo en el ciclo de reloj inmediatamente siguiente, las cargas redundantes en estos registros no dañan.

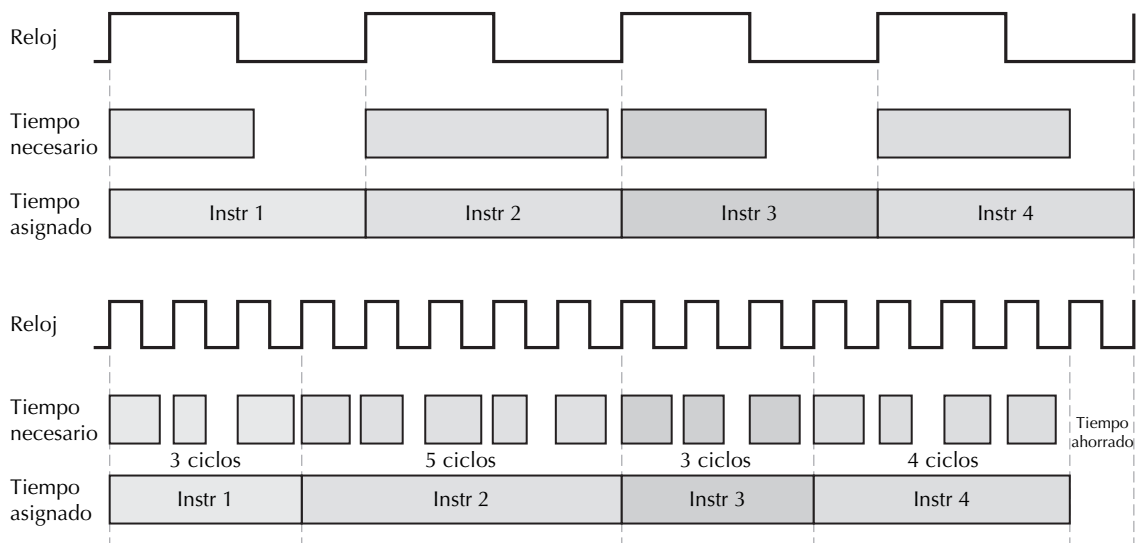


Figura 14.1 Ejecución de instrucción de ciclo sencillo frente a multiciclo.

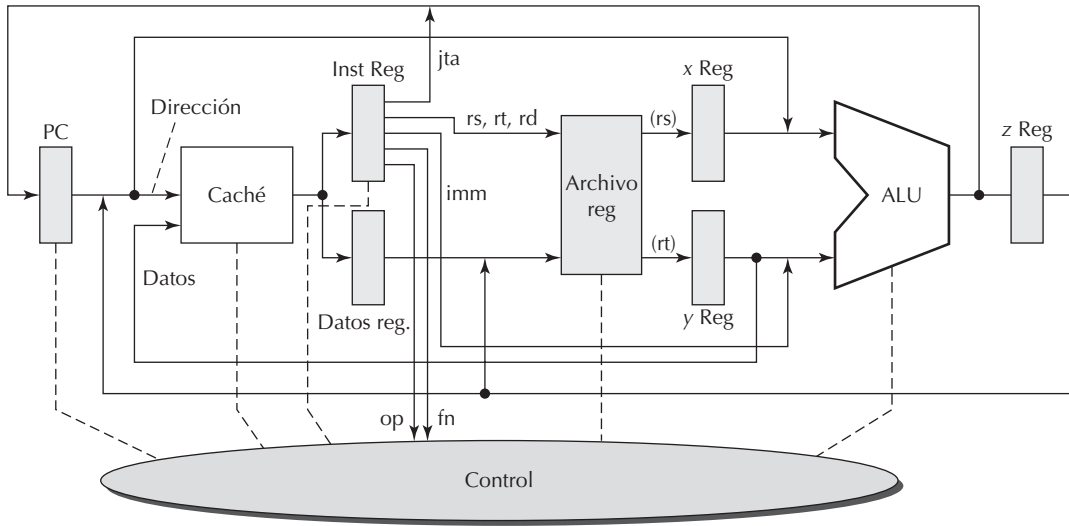


Figura 14.2 Visión abstracta de una unidad de ejecución de instrucción multiciclo para MicroMIPS. Para los nombres de los campos de instrucción vea la figura 13.1.

La ruta de datos en la figura 14.2 es capaz de ejecutar una instrucción cada 3-5 ciclos de reloj; de ahí el nombre de “ruta de datos multiciclo”. La ejecución de todas las instrucciones comienza de la misma forma en el primer ciclo: el contenido de PC se usa para acceder al caché y la palabra recuperada se coloca en el registro de instrucción. Esto se denomina ciclo de *lectura (fetch) de instrucción*. El segundo ciclo se dedica a decodificar la instrucción y también a acceder a los registros *rs* y *rt*. Observe que no toda instrucción requiere dos operandos de los registro y en este punto todavía no se sabe cuál instrucción se ha leído. Sin embargo, leer *rs* y *rt* no daña, incluso si se evidencia que no se necesitan los contenidos de uno o ambos registros. Si la instrucción a mano representa una de las cuatro instrucciones *jump* (*j*, *jr*, *jal*, *syscall*), su ejecución termina en el tercer ciclo al escribir la dirección adecuada en PC. Para el caso de que se trate de una instrucción *branch* (*beq*, *bne*, *bltz*), la condición *branch* se verifica y el valor apropiado se escribirá en PC en el tercer ciclo. Todas las otras instrucciones proceden al cuarto ciclo, donde se completan. Sólo hay una excepción: *lw* requiere de un quinto ciclo para escribir los datos recuperados del caché en un registro.

La figura 14.3 muestra algunos de los detalles de la ruta de datos que se pierden de la versión abstracta de la figura 14.2. Los multiplexores sirven a las mismas funciones que las usadas en la figura 13.2. El multiplexor de tres entradas a la entrada al archivo de registro permite que *rt*, *rd* o *\$31* se usen como el índice del registro de destino en el que se escribirá un resultado. El multiplexor de dos entradas en la parte inferior del archivo de registro permite que los datos leídos del caché o la salida de la ALU se escriban en el registro seleccionado. Lo anterior sirve a la misma función que el mux de tres entradas en el borde derecho de la figura 13.3. La razón para tener una entrada menos aquí es que el PC aumentado ahora se forma por la ALU en lugar de por una unidad separada. Los multiplexores de dos y cuatro entradas cerca del borde derecho de la figura 14.3 corresponden al mux en el bloque *next-address* de la figura 13.3 (figura 13.4). La dirección de la instrucción siguiente a escribirse en PC puede provenir de cinco fuentes posibles. Los primeros dos son los adecuadamente modificados *jta* y *SysCallAddr*, correspondientes a las instrucciones *j* y *syscall*. Se selecciona uno de estos dos valores y se envía a la entrada superior del multiplexor “PC fuente” de cuatro entradas. Las otras tres

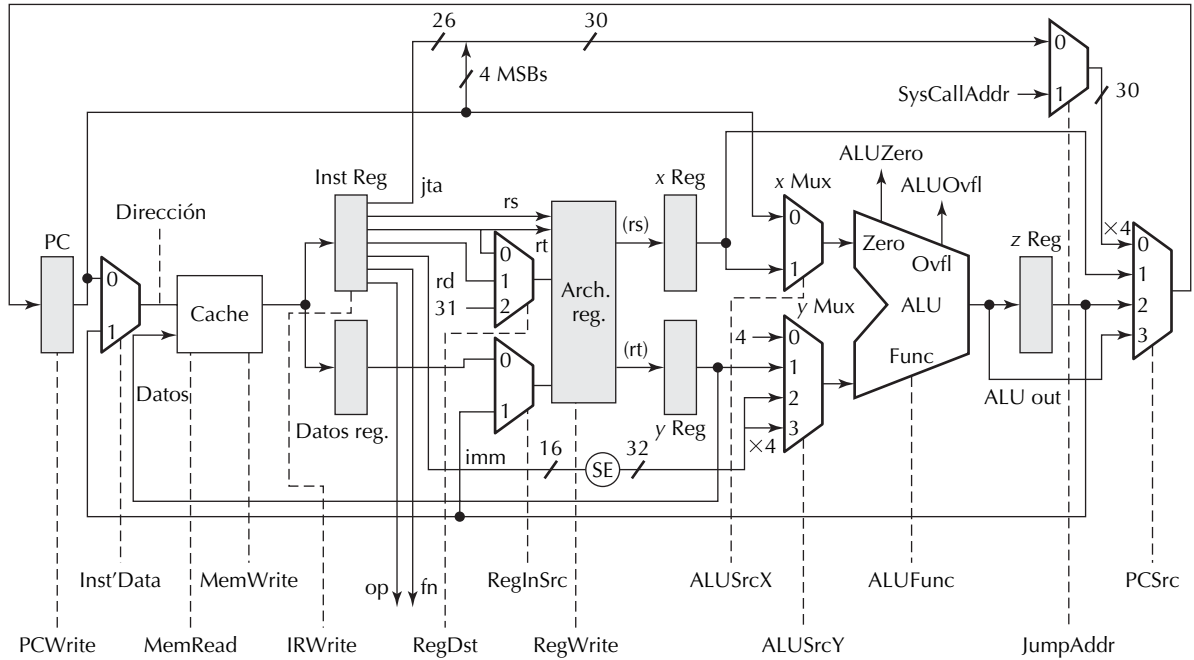


Figura 14.3 Elementos clave de la ruta de datos multiciclos de MicroMIPS.

entradas de este multiplexor son (rs) del registro x , la salida ALU en el ciclo precedente del registro z y la salida ALU en el ciclo actual.

En comparación con la figura 13.3, se agregó un mux para la entrada superior a la ALU (mux x) y se expandió el multiplexor de entrada inferior ALU (mux y) de dos a cuatro entradas. Esto último sucede porque la ALU ahora también debe calcular $(PC) + 4$ y $(PC) + 4 + imm$. Para calcular $(PC) + 4$, el mux x se provee con la señal de control 0 y el mux y con 00. Lo anterior se hace en el primer ciclo de instrucción de ejecución por cada instrucción. Luego, el valor PC incrementado se añade en un ciclo subsecuente a $4 \times imm$ con el uso de los establecimientos de control 0 y 11 para los mux x y y , respectivamente. Note que dos versiones del valor inmediato extendido en signo se pueden usar como la entrada inferior a la ALU: la versión regular, que se necesita para instrucciones como *addi*, y la versión de corrimiento izquierda (multiplicada por 4), que se necesita para lidiar con el *offset* en instrucciones *branch*.

■ 14.2 Ciclo de reloj y señales de control

En el control multiciclo, las acciones necesarias para ejecutar cada instrucción se dividen, y un subconjunto de estas acciones se asigna a cada ciclo de reloj. El periodo de reloj se debe elegir para equilibrar el trabajo realizado en cada ciclo, con la meta de minimizar la cantidad de tiempo de inactividad; el tiempo de inactividad excesivo conduce a pérdida de rendimiento. Como en la sección 13.5, se suponen las latencias siguientes para los pasos básicos en ejecución de instrucción:

Acceso a memoria (leer o escribir)	2 ns
Acceso a registro (leer o escribir)	1 ns
Operación ALU	2 ns

■ **TABLA 14.1** Señales de control para la implementación MicroMIPS multiciclo.

Bloque	Señal de control	0	1	2	3
Contador de programa	JumpAddr	jta	SysCallAddr		
	PCSrc ₁ , PCSrc ₀	Jump addr	x reg	z reg	ALU out
	PCWrite	No escribe	Escribe		
Caché	Inst'Data	PC	z reg		
	MemRead	No lee	Lee		
	MemWrite	No escribe	Escribe		
Archivo de registro	IRWrite	No escribe	Escribe		
	RegWrite	No escribe	Escribe		
	RegDst ₁ , RegDst ₀	rt	rd	\$31	
	RegInSrc	Data reg	z reg		
	ALUSrcX	PC	x reg		
ALU	ALUSrcY ₁ , ALUSrcY ₀	4	y reg	imm	4 × imm
	Add/Sub	Suma	Resta		
	ALUFunc	LogicFn ₁ , LogicFn ₀	AND	OR	NOR
		FnClass ₁ , FnClass ₀	lui	Set less	Aritmética
					Lógica

Por tanto, un ciclo de reloj de 2 ns realizaría cada uno de los pasos básicos de una instrucción en un ciclo de reloj. Esto último conduce a una frecuencia de reloj de 500 MHz. Si los números no incluyen un margen de seguridad, entonces podría necesitarse un periodo de reloj un poco más largo (por decir, 2.5 ns para una frecuencia de reloj de 400 MHz) para acomodar la cabecera agregada de valores almacenados en registros en cada ciclo. Aquí se procede con la suposición de que un periodo de reloj de 2 ns es suficiente.

La tabla 14.1 contiene un listado y definiciones de todas las señales de control que se muestran en la figura 14.3, agrupados por el bloque en el diagrama afectado por la señal. Las primeras tres entradas en la tabla 14.1 se relacionan con el contador de programa. PCSrc selecciona de entre los cuatro valores alternos para cargar en el PC y PCWrite especifica cuándo se modifica PC. Las cuatro fuentes para el nuevo contenido PC son:

- 00 Dirección directa (PC₃₁₋₂₈|jta|00 o SysCallAddr), seleccionado por JumpAddr
- 01 Contenido del registro *x*, que retiene el valor leído de *rs*
- 10 Contenido del registro *z*, que retiene la salida ALU del ciclo precedente
- 11 Salida ALU en el ciclo actual

Observe que en la figura 14.3, la extensión derecha de PC₃₁₋₂₈|jta con 00 se muestra como multiplicación por 4.

Cuatro señales binarias se asocian con la caché. Las señales MemRead y MemWrite se explican por ellas mismas. La señal Inst'Data indica si la memoria se accede leer (*fetch*) una instrucción (dirección proveniente del PC) o para leer/escribir datos (dirección calculada por la ALU en el ciclo precedente). La señal IRWrite, que se postula en el ciclo *fetch* de instrucción, indica que la salida de memoria se escribirá en el registro de instrucción. Observe que el registro de datos no tiene señal de control de escritura. Esto sucede porque su entrada de control de escritura se liga al reloj, ello provoca que cargue la salida de memoria en cada ciclo, incluso en el ciclo de lectura (*fetch*) de la instrucción. Como se observó antes, las cargas redundantes no causan problemas, siempre que los datos cargados en este registro se usen en el ciclo siguiente antes de ser sobrescrito por algo más.

Las señales de control que afectan el archivo de registro son idénticas a las del diseño de ciclo sencillo de la figura 14.3, excepto que RegInSrc es bi en lugar de trivaluada porque el contenido PC incrementado ahora emerge de la ALU.

Finalmente, las señales de control asociadas con la ALU especifican las fuentes de sus dos operandos y la función a realizar. El operando ALU alto puede provenir de PC o del registro x , bajo el control de la señal $ALUSrcX$. El operando ALU inferior tiene cuatro posibles fuentes, a saber:

- 00 La constante 4 para incrementar el contador de programa
- 01 Contenido del registro y que retiene (rt), que se leyó en ciclo precedente
- 10 El campo inmediato de la instrucción, signo extendido a 32 bits
- 11 El campo *offset* de la instrucción, extendido en signo y corrimiento a la izquierda por dos bits

De nuevo, adjuntar el *offset* extendido en signo con 00 a la derecha se muestra como multiplicación por 4. Las señales que comprende $ALUFunc$ tienen los mismos significados que los del diseño de ciclo sencillo (tabla 13.2).

La tabla 14.2 muestra los establecimientos de señal de control en cada ciclo de reloj durante la ejecución de una instrucción. Los primeros dos ciclos son comunes a todas las instrucciones. En el ciclo 1, la instrucción se lee (*fetch*) del caché y se coloca en el registro de instrucción. En virtud de que la ALU está libre en este ciclo, se le usa para incrementar el PC. De esta forma, el valor PC incrementado también estará disponible para sumar al *offset* bifurcado. En el ciclo 2, se leen los registros rs y rt , y sus contenidos se escriben en los registros x y y , respectivamente. Además, la instrucción se decodifica y en el registro z se forma la dirección destino de bifurcación. Aun cuando en la mayoría de los casos la instrucción evidenciará ser algo distinto a *branch*, usar el ciclo inactivo de ALU para precalcular la dirección de bifurcación no daña. Observe que aquí se supone que el ciclo de reloj de 2 ns proporciona suficiente tiempo para lectura de registro y suma de latencia a través de la ALU.

■ **TABLA 14.2** Ciclos de ejecución para la implementación multiciclo de MicroMIPS.

Ciclo de reloj	Instrucción	Operaciones	Establecimientos de señal
1 <i>Fetch</i> e incremento PC	Cualquiera	Lee la instrucción y la escribe en el registro de instrucción; incrementa PC	$Inst'Data = 0$, $MemRead = 1$ $IRWrite = 1$, $ALUSrcX = 0$ $ALUSrcY = 0$, $ALUFunc = '+'$ $PCSrc = 3$, $PCWrite = 1$
2 Decodifica y lectura de registro	Cualquiera	Lee rs y rt en los registros x y y ; calcula la dirección de bifurcación y la salva en el registro z	$ALUSrcX = 0$, $ALUSrcY = 3$ $ALUFunc = '+'$
3 Operación ALU y actualización PC	Tipo ALU	Realiza operación ALU y salva el resultado en el registro z	$ALUSrcX = 1$, $ALUSrcY = 1$ o 2 $ALUFunc$: Varies
	Load/Store	Suma los valores base y <i>offset</i> ; salva en registro z	$ALUSrcX = 1$, $ALUSrcY = 2$ $ALUFunc = '+'$
	Branch	Si $(x\text{ reg}) \neq (y\text{ reg})$, fija PC para dirección destino de bifurcación	$ALUSrcX = 1$, $ALUSrcY = 1$ $ALUFunc = '-'$, $PCSrc = 2$ $PCWrite = ALUZero$ o $ALUZero' \text{ o } ALUOut_{31}$
	Jump	Fija PC a la dirección blanco $jumpa$, $SysCallAddr$ o (rs)	$JumpAddr = 0$ o 1 $PCSrc = 0$ o 1, $PCWrite = 1$
4 Escribir registro o acceso a memoria	Tipo ALU	Escribe reg z en rd	$RegDst = 1$, $RegInSrc = 1$ $RegWrite = 1$
	Load	Lee memoria en reg datos	$Inst'Data = 1$, $MemRead = 1$
	Store	Copia reg y en memoria	$Inst'Data = 1$, $MemWrite = 1$
5 Escribir registro para <i>lw</i>	Load	Copia registro de datos en rt	$RegDst = 0$, $RegInSrc = 0$ $RegWrite = 1$

Al comenzar con el ciclo 3, se conoce la instrucción a mano porque la decodificación se completa en el ciclo 2. Los valores de señal de control en el ciclo 3 dependen de la categoría de instrucción: tipo ALU, *load/store*, *branch* y *jump*. Las últimas dos categorías de instrucciones se completan en el ciclo 3, mientras que otras se mueven al ciclo 4, donde las instrucciones de tipo ALU terminan y se prescriben acciones diferentes para *load* y *store*. Finalmente, *lw* es la única instrucción que necesita el ciclo 5 para su conclusión. Durante este ciclo, el contenido del registro de datos se escribe en el registro *rt*.

14.3 La máquina de control de estado

La unidad de control debe distinguir entre los cinco ciclos del diseño multiciclo y ser capaz de realizar diferentes operaciones dependiendo de la instrucción. Observe que el establecimiento de cualquier señal de control se determina unívocamente si se sabe el tipo de instrucción que se ejecuta y cuál de sus ciclos se encuentra en progreso. La máquina de control de estado lleva la información requerida de estado (*state*) a estado, donde cada estado se asocia con un conjunto de valores particular para las señales de control.

La figura 14.4 muestra los estados de control y las transiciones de estado. La máquina de control de estado se establece en el estado 0 cuando comienza la ejecución de programa. Luego se mueve de estado a estado hasta que una instrucción se completa, en cuyo tiempo regresa al estado 0 para comenzar la ejecución de otra instrucción. Este *ciclo* a través de los estados de la figura 14.4 continúa hasta que se ejecuta una instrucción *syscall* con el número 10 en el registro *\$v0* (vea tabla 7.2). Esta instrucción termina la ejecución del programa.

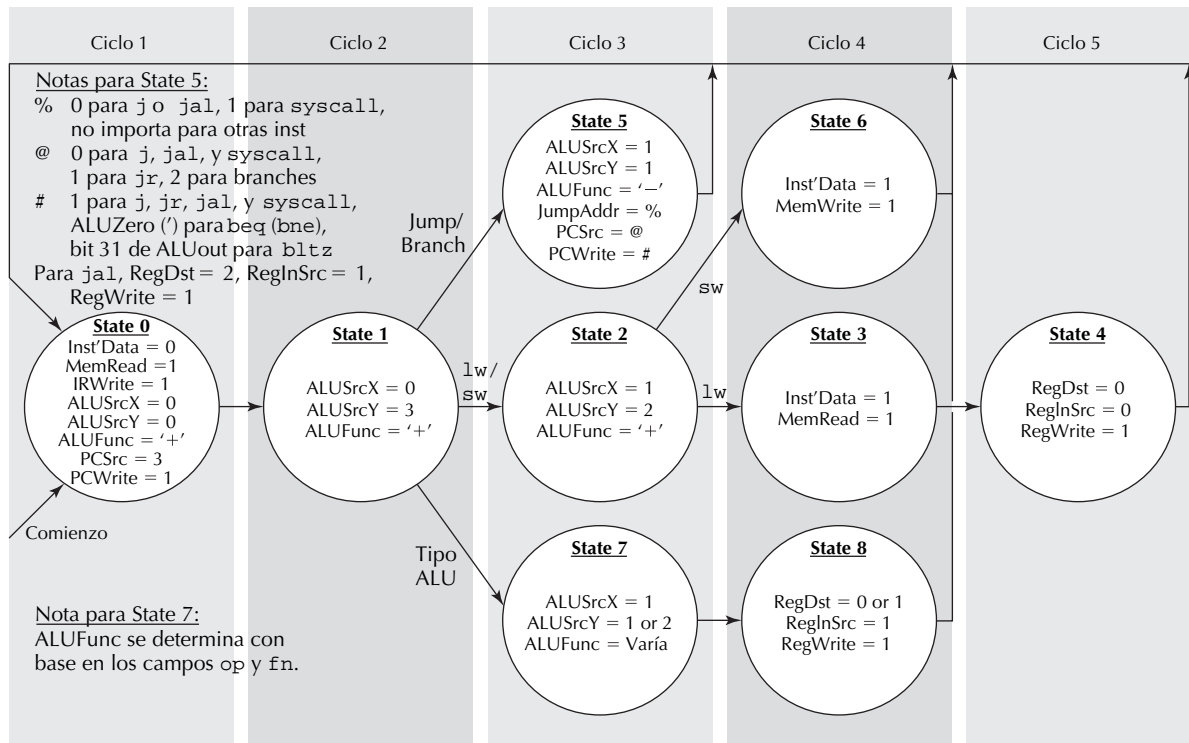


Figura 14.4 La máquina de control de estado para MicroMIPS multiciclo.

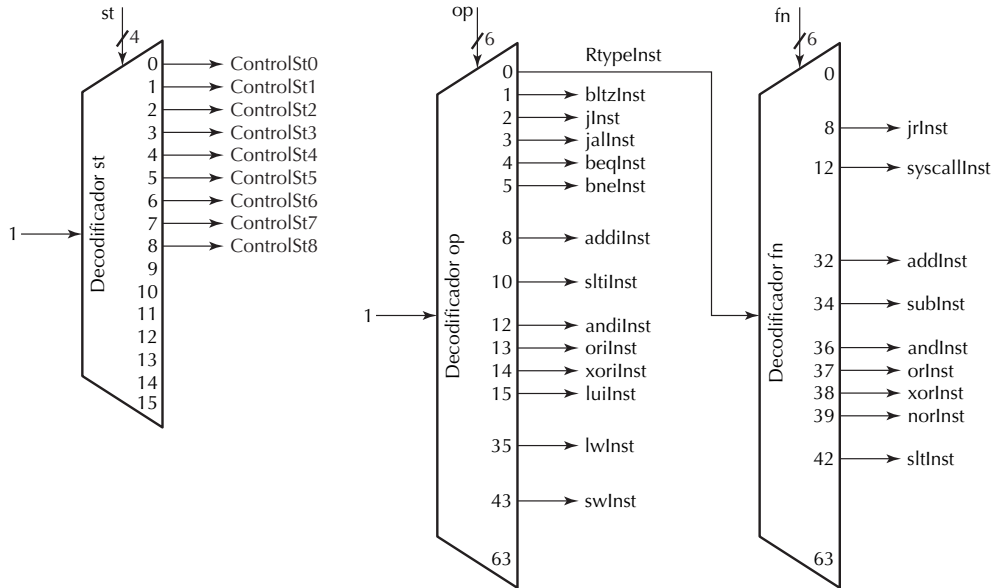


Figura 14.5 Decodificadores de estado e instrucción para MicroMIPS multiciclo.

Las secuencias de control de estado para diversas clases de instrucción MicroMIPS son las siguientes:

Tipo ALU	0, 1, 7, 8
Cargar palabra	0, 1, 2, 3, 4
Almacenar palabra	0, 1, 2, 6
Jump/branch	0, 1, 5

En cada estado, excepto para los estados 5 y 7, los establecimientos de señal de control se determinan de manera única. La información acerca del estado de control actual y la instrucción que se ejecuta se proporcionan en los decodificadores mostrados en la figura 14.5. Observe que la porción del decodificador de instrucción de la figura 14.5 (compuesta de los decodificadores op y fn) es idéntica a la de la figura 13.5, y las salidas de este decodificador se usan para determinar las señales de control de los estados 5 y 7, así como para controlar las transiciones de estado de acuerdo con la figura 14.4. Lo que se añadió es un decodificador de estado que postula la señal ControlSt i siempre que la máquina de control de estado esté en el estado i .

Las expresiones lógicas para las señales de control de la implementación MicroMIPS multiciclo se pueden derivar fácilmente con base en las figuras 14.4 y 14.5. Los ejemplos de señales de control que se determinan de manera única mediante información de estado de control incluyen:

$$\text{ALUSrcX} = \text{ControlSt2} \vee \text{ControlSt5} \vee \text{ControlSt7}$$

$$\text{RegWrite} = \text{ControlSt4} \vee \text{ControlSt8}$$

Los establecimientos de las señales ALUFunc dependen no sólo del estado de control, sino también de la instrucción específica que se ejecuta. Defina un par de señales de control auxiliares:

$$\text{addsubInst} = \text{addInst} \vee \text{subInst} \vee \text{addiInst}$$

$$\text{logicInst} = \text{andInst} \vee \text{orInst} \vee \text{xorInst} \vee \text{norInst} \vee \text{andiInst} \vee \text{oriInst} \vee \text{xoriInst}$$

En consecuencia, las señales de control ALU se pueden establecer como sigue:

$$\begin{aligned} \text{Add/Sub} &= \text{ControlSt5} \vee (\text{ControlSt7} \wedge \text{subInst}) \\ \text{FnClass}_1 &= \text{ControlSt7}' \vee \text{addsubInst} \vee \text{logicInst} \\ \text{FnClass}_0 &= \text{ControlSt7} \wedge (\text{logicInst} \vee \text{sltInst} \vee \text{sltiInst}) \\ \text{LogicFn}_1 &= \text{ControlSt7} \wedge (\text{xorInst} \vee \text{xoriInst} \vee \text{norInst}) \\ \text{LogicFn}_0 &= \text{ControlSt7} \wedge (\text{orInst} \vee \text{oriInst} \vee \text{norInst}) \end{aligned}$$

El control de estado 5 es similar al estado 7, en cuanto que los establecimientos de señal de control para él dependen de la instrucción que se ejecuta y algunas otras condiciones. Por ejemplo:

$$\begin{aligned} \text{JumpAddr} &= \text{syscallInst} \\ \text{PCSrc}_1 &= \text{ControlSt0} \vee \text{ControlSt5} \wedge (\text{beqInst} \vee \text{bneInst} \vee \text{bltzInst}) \\ \text{PCSrc}_0 &= \text{ControlSt0} \vee \text{ControlSt5} \wedge \text{jrInst} \\ \text{PCWrite} &= \text{ControlSt0} \vee \text{ControlSt5} \wedge [\text{jInst} \vee \text{jrInst} \vee \text{jalInst} \vee \text{syscallInst} \vee \\ &\quad (\text{ALUZero} \wedge \text{beqInst}) \vee (\text{ALUZero}' \wedge \text{bneInst}) \vee (\text{ALUout}_{31} \wedge \text{bltzInst})] \end{aligned}$$

Los circuitos de control así derivados serían un poco más simples si los estados 5 y 7 de la máquina de control de estado se expandieron en estados múltiples correspondientes a establecimientos de señal idénticos o similares. Por ejemplo, el estado 5 se podría descomponer en los estados 5b (para bifurcaciones) y 5j (para saltos), o en estados múltiples, uno para cada instrucción diferente. Sin embargo, lo anterior complicaría la máquina de estado de control y el decodificador de estado asociado, de manera que podría ser ineficiente en costo.

14.4 Rendimiento del diseño multiciclo

La implementación MicroMIPS multiciclo discutida en las secciones precedentes también se pueden referir como una implementación multiestado. Esto último está en contraste con la implementación de control de estado sencillo o sin memoria del capítulo 13. El control de ciclo sencillo es sin memoria en el sentido de que la ejecución de cada ciclo comienza de nuevo, y la señales y eventos durante un ciclo se basan sólo en la instrucción actual que se ejecuta; no se afectan por lo que transpira en ciclos previos.

En la sección 13.6 se pudo observar que el MicroMIPS de ciclo sencillo tiene un CPI de 1, en virtud de que una nueva instrucción se ejecuta en cada ciclo de reloj. Para evaluar el rendimiento del MicroMIPS multiciclo, se calcula el CPI promedio con el uso de la misma mezcla de instrucciones como se usó en la sección 13.6. La aportación de cada clase de instrucción al CPI promedio se obtiene al multiplicar su frecuencia por el número de ciclos necesarios para ejecutar instrucciones en dicha clase:

			Aportación a CPI
Tipo R	44%	4 cliclos	1.76
Load	24%	5 cliclos	1.20
Store	12%	4 cliclos	0.48
Branch	18%	3 cliclos	0.54
Jump	2%	3 cliclos	0.06
CPI \cong promedio			4.04

Observe que el número de ciclos para cada clase de instrucción lo determinan los pasos necesarios para su ejecución (tabla 14.2).

Con la tasa de reloj de 500 MHz, derivada al comienzo de la sección 14.2, el CPI de 4.04 corresponde a un rendimiento de $500/4.04 \cong 123.8$ MIPS. Esto es lo mismo que el rendimiento de 125 MIPS de

la implementación de ciclo sencillo derivada en la sección 13.6. Por tanto, las dos implementaciones de MicroMIPS en los capítulos 13 y 14 tienen rendimiento comparable, en parte porque las latencias de instrucción no son muy diferentes unas de otras; la instrucción más lenta tiene una latencia que es $8/5 = 1.6$ veces la de la más rápida. Si las latencias de instrucción hubieran sido más variadas, el diseño multiciclo habría llevado a una ganancia de rendimiento sobre la implementación de ciclo sencillo.

Ejemplo 14.1: Mayor variabilidad en los tiempos de ejecución de instrucción Considere una implementación multiciclo de MicroMIPS++, una máquina es similar a MicroMIPS, excepto que sus instrucciones de tipo R caen en tres categorías.

- a) Las instrucciones de tipo R_a , que constituyen la mitad de todas las instrucciones de tipo R ejecutadas, tardan cuatro ciclos.
- b) Las instrucciones de tipo R_b , que constituyen una cuarta parte de todas las instrucciones de tipo R ejecutadas, tardan seis ciclos.
- c) Las instrucciones de tipo R_c , que constituyen una cuarta parte de todas las instrucciones de tipo R ejecutadas, tardan diez ciclos.

Con la mezcla de instrucciones dada al comienzo de la sección 14.4, y suponiendo que las instrucciones de tipo R más lentas tardarán 16 ns en ejecutarse en una implementación de ciclo sencillo, derive la ventaja en rendimiento de la implementación multiciclo sobre la implementación de ciclo sencillo.

Solución: El tiempo supuesto de ejecución del peor caso de 16 ns conduce a una tasa de reloj de 62.5 MHz y rendimiento de 62.5 MIPS para el diseño de ciclo sencillo. Para el diseño multiciclo, el único cambio en el cálculo CPI promedio es que la aportación de las instrucciones de tipo R al CPI promedio aumenta de 1.76 a $0.22 \times 4 + 0.11 \times 6 + 0.11 \times 10 = 2.64$. Esto último eleva el CPI promedio de 4.04 a 4.92. Por tanto, el rendimiento de MicroMIPS++ multiciclo se convierte en $500/4.92 \cong 101.6$ MIPS. El factor de mejora de rendimiento del diseño multiciclo sobre la implementación de ciclo sencillo es, en consecuencia, $101.6/62.5 = 1.63$. Observe que la inclusión de instrucciones de tipo R más complejas que tardan seis y diez ciclos en ejecutarse tiene un efecto relativamente pequeño sobre el rendimiento del diseño multiciclo (de 123.8 MIPS a 101.6 MIPS, para una reducción de aproximadamente 18%), mientras que recortan a la mitad el rendimiento del diseño de ciclo sencillo.

14.5 Microprogramación

La máquina de control de estado de la figura 14.4 se parece a un programa que tiene instrucciones (los estados), bifurcación y *ciclos*. A tal programa en hardware se le llama *microprograma* y a sus pasos básicos, *microinstrucción*. Dentro de cada una de éstas se prescriben diferentes acciones, como postulación de la señal de control MemRead o el establecimiento de ALUFunc a “+”. Cada una de tales acciones representa una *microorden*. En lugar de implementar la máquina de estado de control en hardware a la medida, se pueden almacenar microinstrucciones en ubicaciones de un ROM de control, y leer (*fetch*), así como ejecutar una secuencia de microinstrucciones para cada instrucción en lenguaje de máquina. De este modo, en la misma forma en la que un programa o procedimiento se descompone en instrucciones de máquina, una instrucción de máquina, a su vez, se descompone en una secuencia de microinstrucciones. Por ende, cada microinstrucción define un paso en la ejecución de una instrucción en lenguaje de máquina.

La implementación de control basada en ROM tiene muchas ventajas. Hace el hardware más simple, más regular y menos dependiente de los detalles de la arquitectura del conjunto de instrucciones, de modo que el mismo hardware se puede usar para diferentes propósitos sólo con la modificación de

los contenidos ROM. Además, conforme el diseño de hardware progresa desde la planificación hasta la implementación, es posible corregir errores y omisiones al cambiar el microprograma, en oposición a un costoso rediseño y nueva fabricación de circuitos integrados. Finalmente, los conjuntos de instrucciones se pueden afinar y agregar nuevas instrucciones, al cambiar y expandir el microprograma. Una máquina con este tipo de control está *microprogramada*. Diseñar una secuencia adecuada de microinstrucciones para realizar una arquitectura de conjunto de instrucciones particular representa *microprogramar*. Si el microprograma se puede modificar fácilmente, incluso por el usuario, la máquina es *microprogramable*.

¿Existen inconvenientes para la microprogramación? En otras palabras, ¿la flexibilidad ofrecida por la implementación de control microprogramado viene a un costo? El inconveniente principal es menor rapidez de la que es alcanzable con la implementación de control *alambrado* (*hardwired*). Con la implementación microprogramada, la ejecución de una instrucción MicroMIPS requiere 3-5 accesos ROM para leer (*fetch*) las microinstrucciones correspondientes a los estados de la figura 14.4. Después de que cada microinstrucción se lee y coloca en un *registro de microinstrucción*, se debe asignar tiempo suficiente para que todas las señales se establezcan y sucedan las acciones (como lectura o escritura de memoria). Por tanto, todas las latencias asociadas con la lectura de registro, acceso de memoria y operación ALU aún están en efecto; encima de éstas, algunos retardos de compuertas necesarias para la generación de señal de control se sustituyen por un retardo de acceso ROM que es mucho más largo.

El diseño de una unidad de control microprogramado comienza el diseño de un formato de microinstrucción adecuado. Para MicroMIPS, se puede usar el formato de microinstrucción de 22 bits que se muestra en la figura 14.6. Excepto los dos bits de la extrema derecha denominados *Sequence control* (secuencia de control), los bits en la microinstrucción están en correspondencia uno a uno con las señales de control en la ruta de datos multiciclo de la figura 14.3. En consecuencia, cada microinstrucción define explícitamente el establecimiento para cada una de las señales de control.

El campo *secuencia de control* de dos bits permite el control de la secuenciación de microinstrucciones en la misma forma que *PC control* afecta la secuenciación de las instrucciones en lenguaje de máquina. La figura 14.7 muestra que existen cuatro opciones para elegir la siguiente microinstrucción. La opción 0 es avanzar a la siguiente microinstrucción en secuencia mediante el incremento del contador de microprograma. Las opciones 1 y 2 permiten que ocurra bifurcación dependiendo del campo *opcode* en la instrucción de máquina que se ejecuta. La opción 3 es ir a la microinstrucción 0 correspondiente al estado 0 en la figura 14.4. Esto último inicia la fase *fetch* para la siguiente instrucción de máquina. Cada una de las dos tablas de transferencia (*dispatch*) traduce el *opcode* en una dirección de microinstrucción. La tabla de transferencia 1 corresponde a la bifurcación multivía al ir del ciclo 2 al ciclo 3 en la figura 14.4. La tabla de transferencia 2 implementa la bifurcación entre los ciclos 3 y 4. De manera co-

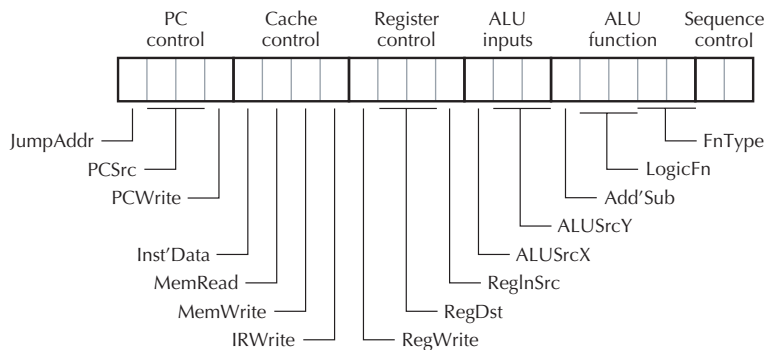


Figura 14.6 Posible formato de microinstrucción de 22 bits para MicroMIPS.

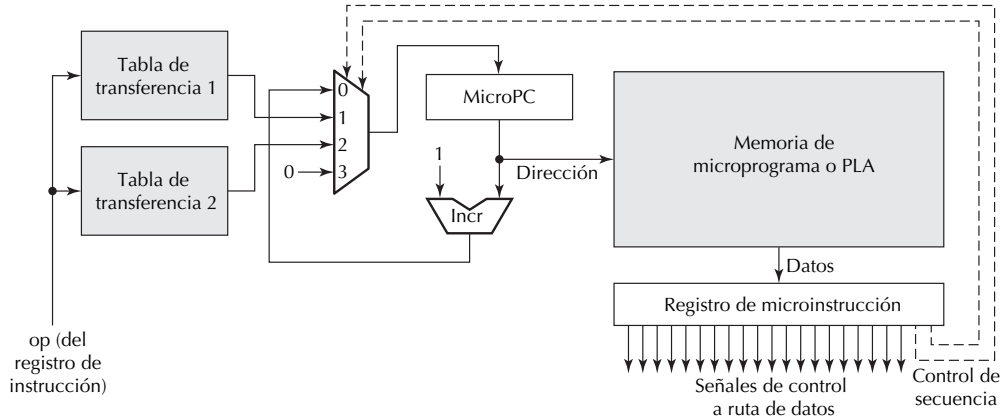


Figura 14.7 Unidad de control microprogramado para MicroMIPS.

■ **TABLA 14.3** Valores de campo microinstrucción y sus nombres simbólicos: el valor por defecto para cada campo no especificado es el patrón de bits todos 0.

Nombre de campo		Posibles valores de campo y sus nombres simbólicos			
PC control	0001	1001	x011	x101	x111
	PCjump	PCsyscall	PCjreg	PCbranch	PCnext
Cache control	0101	1010	1100		
	CacheFetch	CacheStore	CacheLoad		
Register control	1000	1001	1011	1101	
	rt ← Data	rt ← z	rd ← z	\$31 ← PC	
ALU inputs*	000	011	101	110	
	PC ⊗ 4	PC ⊗ 4imm	x ⊗ y	x ⊗ imm	
ALU function*	0xx10	1xx01	1xx10	x0011	x0111
	+	<	-	^	∨
	x1011	x1111	xxx00		
	⊕	~∨	lui		
Sequence control	01	10	11		
	μPCdisp1	μPCdisp2	μPCfetch		

*Nota: El símbolo operador ⊗ representa cualquiera de las funciones ALU definidas anteriormente (excepto para “lui”).

lectiva, las dos tablas de transferencia permiten dos bifurcaciones multivía dependientes de instrucción en el curso de la ejecución de instrucción; por tanto, se habilita la compartición de microinstrucciones dentro de clases de instrucciones que siguen pasos más o menos similares en sus ejecuciones.

Ahora se puede proceder a escribir el microprograma para la implementación MicroMIPS multiciclo. Con el propósito de hacer más legible el microprograma, se usan los nombres simbólicos que se muestran en la tabla 14.3 con el fin de designar combinaciones de valores bit en los diversos campos de la microinstrucción.

Como ejemplo, la microinstrucción

x111 0101 0000 000 0xx10 00

se escribe en forma simbólica mucho más legible como:

PCnext, CacheFetch, PC + 4

Observe que dos de los campos (*register control* y *sequence control*), que tienen los establecimientos por defecto todos 0, no aparecen en esta representación simbólica. Estos establecimientos por defecto sólo especifican que no ocurre escritura de registro y que la siguiente microinstrucción se ejecuta a continuación. Además, los establecimientos de los dos campos *ALU inputs* y *ALU function* se combinan en una expresión que identifica las entradas aplicadas a la ALU y la operación que realiza mucho más claramente.

Con base en la notación de la tabla 14.3, en la figura 14.8 se muestra el microprograma completo para la implementación de MicroMIPS multiciclo. Observe que cada línea representa una microinstrucción y que a las microinstrucciones cuya etiqueta termina en 1 (2) se llega desde la tabla de transferencia 1 (2) de la figura 14.7.

El microprograma de la figura 14.8, que consta de 37 microinstrucciones, define por completo la operación de hardware MicroMIPS para ejecución de instrucción. Si la microinstrucción superior con la etiqueta *fetch* se almacena en la dirección ROM 0, entonces arrancar la máquina con el μ PC limpiado a 0 provocará que la ejecución del programa comience en la instrucción especificada en PC. Por tanto, la parte del proceso de carga inicial (*booting*) para MicroMIPS consiste en limpiar (*clear*) μ PC a 0 y establecer PC a la dirección de la rutina de sistema que inicializa la máquina.

```

fetch:      PCnext, CacheFetch, PC + 4          # State 0 (start)
           PC + 4imm,  $\mu$ PCdisp1                # State 1
lui1:      lui(imm)                             # State 7lui
           rt  $\leftarrow$  z,  $\mu$ PCfetch             # State 8lui
add1:      x + y                                # State 7add
           rd  $\leftarrow$  z,  $\mu$ PCfetch             # State 8add
sub1:      x - y                                # State 7sub
           rd  $\leftarrow$  z,  $\mu$ PCfetch             # State 8sub
slt1:      x < y                                # State 7slt
           rd  $\leftarrow$  z,  $\mu$ PCfetch             # State 8slt
addi1:     x + imm                             # State 7addi
           rt  $\leftarrow$  z,  $\mu$ PCfetch             # State 8addi
slti1:     x < imm                             # State 7slti
           rt  $\leftarrow$  z,  $\mu$ PCfetch             # State 8slti
andi1:     x  $\wedge$  y                             # State 7andi
           rd  $\leftarrow$  z,  $\mu$ PCfetch             # State 8andi
or1:       x  $\vee$  y                               # State 7or
           rd  $\leftarrow$  z,  $\mu$ PCfetch             # State 8or
xor1:      x  $\oplus$  y                             # State 7xor
           rd  $\leftarrow$  z,  $\mu$ PCfetch             # State 8xor
nor1:      x  $\sim \vee$  y                          # State 7nor
           rd  $\leftarrow$  z,  $\mu$ PCfetch             # State 8nor
andi1:     x  $\wedge$  imm                          # State 7andi
           rt  $\leftarrow$  z,  $\mu$ PCfetch             # State 8andi
ori1:      x  $\vee$  imm                          # State 7ori
           rt  $\leftarrow$  z,  $\mu$ PCfetch             # State 8ori
xori:      x  $\oplus$  imm                          # State 7xori
           rt  $\leftarrow$  z,  $\mu$ PCfetch             # State 8xori
lwsw1:     x + imm,  $\mu$ PCdisp2                  # State 2
lw2:       CacheLoad                          # State 3
           rt  $\leftarrow$  Data,  $\mu$ PCfetch         # State 4
sw2:       CacheStore,  $\mu$ PCfetch              # State 6
jl1:       PCjump,  $\mu$ PCfetch                  # State 5j
jrl1:      PCjreg,  $\mu$ PCfetch                   # State 5jr
branch1:   PCbranch,  $\mu$ PCfetch                # State 5branch
jall:      PCjump, $31  $\leftarrow$  PC,  $\mu$ PCfetch    # State 5jal
syscall1:  PCSyscall,  $\mu$ PCfetch               # State 5syscall

```

Figura 14.8 Microprograma completo para MicroMIPS. Los comentarios a la derecha muestran que cada microinstrucción corresponde a un estado (*state*) o subestado en la máquina de estado de control de la figura 14.4.

regregl:	$x \otimes y$	# State 7regreg
	$rd \leftarrow z, \mu PCfetch$	# State 8regreg
regimm1:	$x \otimes imm$	# State 7regimm
	$rt \leftarrow z, \mu PCfetch$	# State 8regimm

Figura 14.9 Microinstrucciones MicroMIPS alternas que representan los estados 7 y 8 en la máquina de estado de control de la figura 14.4.

Observe que existe gran cantidad de repetición en el microprograma de la figura 14.8. Esto último corresponde a los subestados de los estados 7 y 8 de la máquina de control de esta de la figura 14.4. Si el código de función de cinco bits, que ahora forma parte de la microinstrucción, se proporciona directamente a la ALU por un decodificador separado, sólo se necesitarán dos microinstrucciones para cada uno de los estados 7 y 8 de la máquina de control de estado. Los cambios se muestran en la figura 14.9, donde se sugiere que el microprograma completo ahora consiste de 15 microinstrucciones, y cada microinstrucción tiene 17 bits de ancho (el campo ALU function de cinco bits se elimina de la microinstrucción de 22 bits de la figura 14.6).

Todavía es posible mayor reducción en el número de microinstrucciones (vea los problemas al final de este capítulo para algunos ejemplos).

También es posible reducir todavía más el ancho de las microinstrucciones, mediante la codificación más eficiente de los valores de señal de control. Observe que el formato de microinstrucción de la figura 14.6 retiene un bit por cada una de las 20 señales de control en la ruta de datos de la figura 14.3, más dos bits para controlar la secuenciación de microinstrucciones. Tal enfoque conduce a *microinstrucciones horizontales*. Al remitirse a la tabla 14.3, se nota que el campo caché control de cuatro bits puede retener sólo uno de cuatro posibles patrones de bits: el patrón de bits por defecto 0000 y los tres patrones que se mencionan en la tabla 14.3. Estas cuatro posibilidades se pueden codificar eficientemente en dos bits; por tanto reduce el ancho de microinstrucción por dos bits. Todos los otros campos, excepto para control de secuencia, se pueden compactar de manera similar. Tales codificaciones compactas requieren el uso de decodificadores para derivar los valores reales de señal de control a partir de sus formas codificadas en microinstrucciones. Las microinstrucciones en que las combinaciones de valor de señal están codificadas de manera compacta se conocen como *microinstrucciones verticales*. Para el caso extremo de una codificación vertical, cada microinstrucción especifica una sola microoperación usando un formato que es muy similar al de una instrucción en lenguaje de máquina. En este contexto, el diseñador tiene una serie de opciones, desde las meramente horizontales hasta el formato extremo vertical. Los formatos de microinstrucciones cercanos a las meramente horizontales son más rápidas (porque no necesitan mucha decodificación) y permiten operaciones concurrentes entre los componentes de la ruta de datos.

■ 14.6 Excepciones

El diagrama de estado de control de la figura 14.4 y su alambrado asociado o implementación de control microprogramada, especifica el comportamiento del hardware MicroMIPS en el curso normal de ejecución de instrucción. En tanto nada inusual ocurra, las instrucciones se ejecutan una tras otra y el efecto pretendido del programa se observa en el registro actualizado y en los contenidos de memoria. Sin embargo, las cosas pueden y de hecho van mal dentro de la máquina. Los siguientes son algunos de los problemas, o *excepciones*, con las que se debe lidiar:

- La operación ALU conduce a desbordamiento (se obtiene un resultado incorrecto).
- El campo *opcode* retiene un patrón que no representa una operación legal.
- El verificador de código de detección de error de caché estima el ingreso de una palabra inválida.
- El sensor o monitor señalan una condición peligrosa (por ejemplo, sobrecalentamiento).

Una forma usual de lidiar con tales excepciones es forzar la transferencia de control inmediato a una rutina de sistema operativo conocida como *manipulador de excepción*. Esta rutina inicia acción reparadora para corregir o superar el problema o, en el extremo, termina la ejecución del programa para evitar mayor corrupción de datos o daño a los componentes del sistema.

Además de las excepciones, provocadas por eventos internos al CPU de la máquina, la unidad de control debe luchar con *interrupciones*, que se definen como eventos externos que deben solucionarse pronto. En tanto que las excepciones con frecuencia son eventos indeseables, las interrupciones pueden deberse a conclusión anticipada de tareas por dispositivos de entrada/salida, notificación de disponibilidad de datos de los sensores, llegada de mensajes en una red, etcétera. Sin embargo, la reacción de un procesador ante una interrupción es bastante similar a su manejo de excepciones. En consecuencia, en lo que sigue, el enfoque será sobre las excepciones de desbordamiento y operación ilegal para demostrar los procedimientos y mecanismos requeridos. Las interrupciones se discutirán más profundamente en el capítulo 24. La figura 14.10 muestra cómo se pueden incorporar las excepciones de desbordamiento y operación ilegal en la máquina de estado de control para la implementación MicroMIPS multiciclo.

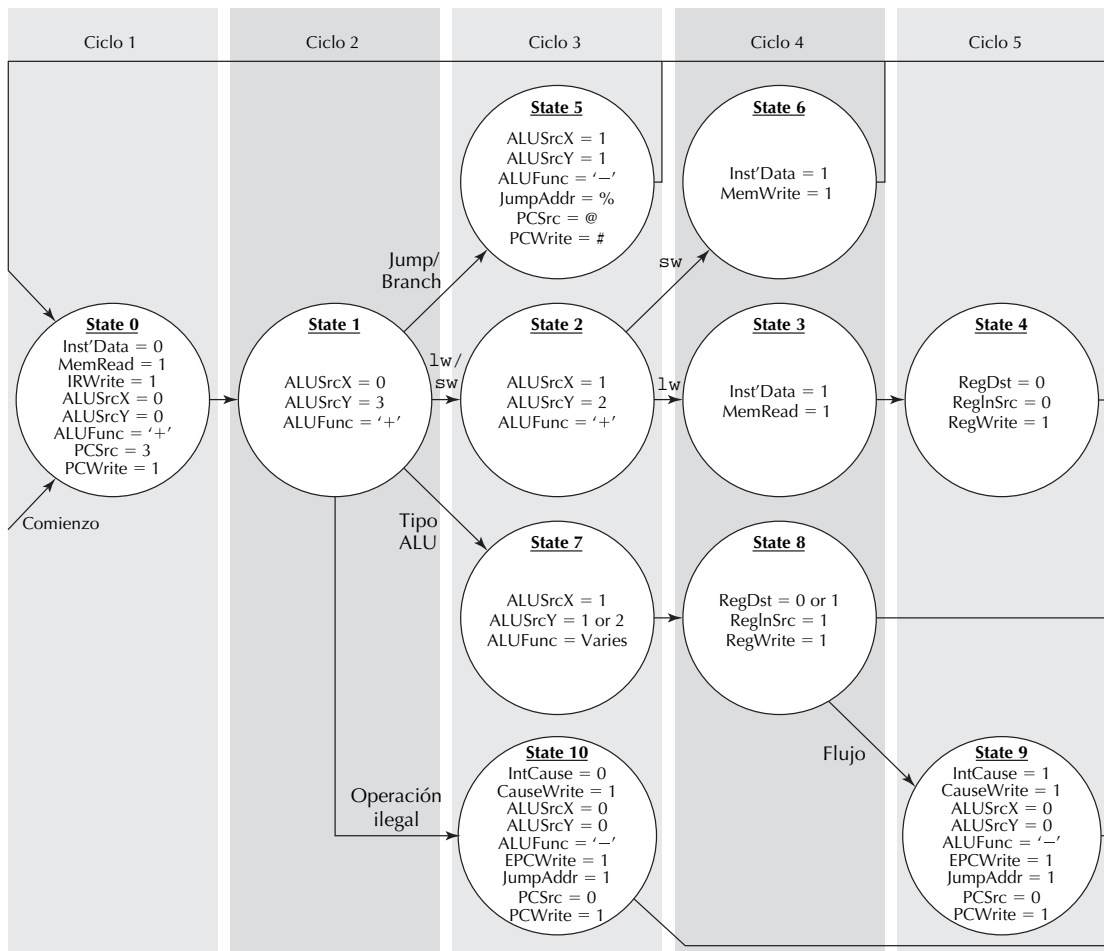


Figura 14.10 Los estados de excepción 9 y 10 se agregan a la máquina de estado de control.

En el estado 8 se observa un desbordamiento aritmético, seguido de la operación ALU en el estado 7. Postular la señal de salida *overflow* de la ALU fuerza a la máquina de estado al estado especial 9. En este sentido la causa de la excepción se puede determinar más tarde mediante la rutina de manejo de excepción, el registro *Cause* se fija a 1 (un código para desbordamiento), el valor actual del contador de programa, menos 4 (para nulificar su avance a la siguiente instrucción), se salva en el registro del contador de programa de excepción (EPC) y el control se transfiere a la dirección *SysCallAddr*, punto de entrada de una rutina de sistema operativo. Para la colocación de los registros *Cause* y EPC en el hardware de la máquina, remítase a la figura 5.1.

En el estado 1 se detecta una operación ilegal en el campo *opcode*, donde se decodifica la instrucción. Una señal correspondiente se postula mediante el circuito de detección, que entonces fuerza la máquina de estado de control en el estado especial 10. De nuevo se salvan un código para causa de la excepción, y la dirección de la instrucción actual que conduce a ella, antes de que el control se transfiera al manipulador de excepción.

Los dos ejemplos discutidos muestran el procedimiento general para lidiar con una excepción o interrupción. En el registro *Cause* se salva un código que muestra la causa de la excepción o interrupción, para el beneficio del sistema operativo. La dirección de la instrucción actual que se ejecuta se salva en el registro EPC de modo que la ejecución del programa se pueda reanudar desde dicho punto una vez que el problema se resuelve. Una instrucción *mfc0* (similar a *mfc1* de la tabla 12.1) permite examinar los contenidos de los registros en el Coprocesador 0. Regresar de la rutina de manipulación de excepción al programa interrumpido es similar a regresar de un procedimiento, como se discute en la sección 6.1.

PROBLEMAS

14.1 Detalles de ruta de datos multiciclo

- Etiquete cada una de las líneas en la figura 14.2 con el número de señales binarias que representa.
- Repita la parte *a)* para la figura 14.3.

14.2 Opciones de selección de ruta de datos

Existen tres pares de multiplexores relacionados en la figura 14.3. En orden de izquierda a derecha, el primer par alimenta el archivo de registro, el segundo suministra operandos a la ALU y el tercero, ubicado cerca del borde derecho del diagrama, selecciona la entrada PC.

- Los multiplexores que alimentan el archivo de registro tienen tres y dos establecimientos, respectivamente; por ende, existen seis combinaciones cuando los establecimientos se toman en conjunto. Para cada una de estas combinaciones, indique si alguna vez se usa y, si es así, para ejecutar cuál(es) instrucción(es) MicroMIPS.
- Repita la parte *a)* para el par de multiplexores que proporcionan los operandos ALU (ocho combinaciones).

- Repita la parte *a)* para el par de multiplexores que seleccionan la entrada PC (ocho combinaciones).

14.3 Extensión de la ruta de datos multiciclo

Sugiera algunos cambios simples (mientras más simples, mejor) en la ruta de datos multiciclo de la figura 14.3, de modo que las siguientes instrucciones se puedan incluir en el conjunto de instrucciones de la máquina:

- Cargar byte (1b).
- Cargar byte sin signo (1bu).
- Almacenar byte (sb).

14.4 Adición de otras instrucciones

Si se desea, ciertas instrucciones se pueden agregar al conjunto de instrucciones de MicroMIPS. Considere las siguientes seudoinstrucciones de la tabla 7.1 y suponga que se quiere incluirlas en MicroMIPS como instrucciones regulares. Para cada caso, elija una codificación adecuada para la instrucción y especifique todas las

modificaciones requeridas en la ruta de datos multiciclo y circuitos de control asociados. Asegúrese de que las codificaciones elegidas no están en conflicto con otras instrucciones MicroMIPS mencionadas en las tablas 6.2 y 12.1.

- a) Mover (move).
- b) Cargar inmediato (li).
- c) Valor absoluto (abs).
- d) Negar (neg).
- e) No (not).
- f) Bifurcación menor que (blt).

14.5 Máquina de estado de control

En la máquina de estado de control de la figura 14.4, el estado 5 no especifica por completo los valores a usar por las señales de control; en vez de ello, suponga el uso de circuitos externos para derivar los valores de señal de control en concordancia con las notas en la esquina superior izquierda de la figura 14.4. Divida el estado 5 en un número mínimo de subestados, llamados 5a, 5b, etc., de modo que dentro de cada subestado todos los valores de señal de control estén determinados (como es el caso para todos los otros estados, excepto para los estados 7 y 8).

14.6 Máquina de control de estado

En la máquina de control de estado de la figura 14.4, los estados 7 y 8 no especifican por completo los valores a usar por todas las señales de control. La especificaciones incompletas, a resolver mediante circuitos de control externos, constan de los valores asignados a las señales de control ALUSrcY y ALUFunc en el estado 7 y RegDst en el estado 8. Demuestre que, si se ignora ALUFunc, todavía se debe determinar de manera externa, las especificaciones de ALUSrcY y RegDst se pueden crear por completo con el uso de dos pares de estados: 7a/8a y 7b/8b.

14.7 Decodificación de instrucción en MicroMIPS

El uso de dos decodificadores 6 a 64 en la figura 14.5 parece un desperdicio, pues sólo es útil un pequeño subconjunto de las 64 salidas en cada decodificador. Demuestre cómo se puede sustituir cada uno de los decodificadores 6 a 64 de la figura 14.5 por un decodificador 4

a 64 y una pequeña cantidad de circuitos lógicos adicionales (mientras más pequeña, mejor). *Sugerencia:* En el decodificador op de la figura 14.5, son útiles la mayoría de las primeras 16 salidas, pero sólo se usan dos de las 48 restantes.

14.8 Señales de control para MicroMIPS multiciclo

Las expresiones lógicas para un número de las 20 señales de control, que se muestran en la figura 14.3, se derivaron en la sección 14.3 con base en las salidas de varios circuitos decodificadores. Derive expresiones lógicas para todas las señales restantes, con la meta de simplificar y compartir componentes de circuito, en la medida posible.

14.9 Rendimiento de control multiciclo

En el ejemplo 14.1, sean f_a , f_b y f_c las frecuencias relativas de las instrucciones tipo R_a , R_b y R_c , respectivamente, con $f_a + f_b + f_c = 1$.

- a) Encuentre una relación entre las tres frecuencias relativas si el diseño multiciclo será 1.8 veces más rápido que el diseño de ciclo sencillo.
- b) Calcule las frecuencias reales del resultado de la parte a) suponiendo que $f_b = f_c$.
- c) ¿Cuál es el factor de aceleración máximo del diseño multiciclo relativo al diseño de ciclo sencillo sobre los posibles valores para las tres frecuencias relativas?

14.10 Rendimiento de control multiciclo

Un conjunto de instrucción está compuesto de h diferentes clases de instrucciones, donde el tiempo de ejecución de las instrucciones de la clase i es $3 + i$ ns, $1 \leq i \leq h$. El control de ciclo sencillo claramente implica un ciclo de reloj de $3 + h$ ns. Considere una implementación de control multiciclo con un ciclo de reloj de q ns y suponga que las instrucciones de la clase i se pueden ejecutar entonces en $3 + i$ ciclos de reloj; esto es, ignore cualquier encabezado asociado con el control multiciclo.

- a) Derive la ventaja de rendimiento (factor de aceleración) del control multiciclo en relación con el control de ciclo sencillo, si supone que las diversas clases de instrucciones se usan con la misma frecuencia.
- b) Demuestre que el beneficio de rendimiento derivado en la parte a) representa una función creciente de h ;

por tanto, pruebe que cualquier factor de aceleración es posible para una h adecuadamente grande.

- c) Repita la parte a) para cuando la frecuencia relativa de las instrucciones de la clase i es proporcional a $1/i$.
- d) ¿Cómo varía el beneficio de rendimiento de la parte c) con h , y el factor de aceleración crece indefinidamente con h creciente, como fue el caso en la parte b)?

14.11 Rendimiento de control multiciclo

Repita el problema 14.10, pero esta vez suponga un ciclo de reloj de 2 ns para el diseño multiciclo, lo cual lleva a que las instrucciones de la clase i requieran $\lceil (3 + i)/2 \rceil$ ciclos de reloj.

14.12 Control de ciclo sencillo frente a multiciclo

- a) Discuta las condiciones en que una implementación multiciclo de control sería inferior a la implementación de ciclo sencillo.
- b) ¿Cómo la variación en la latencia de acceso a memoria afecta el rendimiento de la implementación de control de ciclo sencillo frente a multiciclo, si supone que no cambian el número de ciclos y las acciones realizadas dentro de cada ciclo?
- c) Discuta las implicaciones de sus respuestas a las partes a) y b) en el caso específico de MicroMIPS.

14.13 Microinstrucciones

- a) Construya una tabla con seis columnas y 37 hileras que contenga los contenidos de campo binario para el microprograma que se muestra en la figura 14.8.
- b) Repita la parte a), pero suponga que el microprograma de la figura 14.8 se modificó de acuerdo con la figura 14.9 y la discusión asociada.

14.14 Control microprogramado

- a) ¿Cómo cambiaría el microprograma de la figura 14.8 si el hardware controlador de la figura 14.7 contuviera sólo una tabla de transferencia en lugar de dos?
- b) ¿Qué cambios se necesitan en la figura 14.7, si se incluyese una tercera tabla de transferencias?
- c) Argumente el hecho de que el cambio en la parte b) no ofrece ventajas con respecto al microprograma de la figura 14.8.
- d) ¿Bajo qué circunstancias el cambio en la parte b) sería desventajoso?

14.15 Microprogramación

- a) En conexión con la figura 14.8, se observó que una simple modificación permite tener dos subestados para cada uno de los estados 7 y 8, a saber, 7regreg, 7regimm, 8regreg, 8regimm. ¿Es posible eliminar estos subestados juntos y tener sólo una microinstrucción para cada uno de los estados 7 y 8?
- b) ¿Qué permite la fusión de todos los subestados del estado 5, correspondientes a las últimas cinco microinstrucciones en el microprograma de la figura 14.8, en una sola microinstrucción?

14.16 Formatos de microinstrucción

La discusión al final de la sección 14.5 indica que el ancho de una microinstrucción se reduce si se usa una codificación más compacta para las combinaciones válidas de valores de señal en cada campo.

- a) Si supone que se conservan los mismos seis campos de la figura 14.6 en la microinstrucción, y que la codificación más compacta se usa para cada campo, determine el ancho de la microinstrucción.
- b) Demuestre que es posible reducir aún más el ancho de la microinstrucción mediante la combinación de dos o más de sus campos en un solo campo. Argumente que los ahorros logrados en el ancho de microinstrucción no justifican la complejidad de la decodificación añadida y el retardo asociado.

14.17 Manipulación de excepción

Hasta encontrar una excepción, se podrían salvar los contenidos actuales de PC, en lugar de $(PC) - 4$, en el contador de programa de excepción, ello propicia que el sistema operativo suponga la dirección de la instrucción ofensiva. ¿Esta modificación simplificaría la ruta de datos MicroMIPS multiciclo o su máquina de estado de control asociada? Justifique completamente su respuesta.

14.18 Manipulación de excepción

Suponga que instrucciones y datos se almacenan en la caché de la MicroMIPS multiciclo con el uso de un código detector de error. Cada vez que se accede a la memoria caché, un circuito especial de detección de error verifica la validez de la palabra codificada y postula la señal CacheError si se trata de un código de palabra in-

válido. Agregue estados de excepción a la figura 14.10 para lidiar con un error detectado en:

a) Palabra de instrucción.

b) Palabra de datos de memoria.

a) Diseñe una unidad de control alambrada para URISC.

b) Describa una implementación microprogramada de URISC y proporcione el microprograma completo con el uso de una notación adecuada.

14.19 URISC

Considere el procesador URISC descrito en la sección 8.6 [Mava88].

REFERENCIAS Y LECTURAS SUGERIDAS

- [Andr80] Andrews, M., *Principles of Firmware Engineering in Microprogram Control*, Computer Science Press, 1980.
- [Bane82] Banerji, D. K. y J. Raymond, *Elements of Microprogramming*, Prentice-Hall, 1982.
- [Mava88] Mavaddat, F., and B. Parhami, "URISC: The Ultimate Reduced Instruction Set Computer", *Int. J. Electrical Engineering Education*, vol. 25, pp. 327-334, 1988.
- [Patt98] Patterson, D. A. y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2a. ed., 1998.

RUTAS DE DATOS ENCAUZADAS

“No hay camino sencillo en torno a la penalización por retardo de bifurcación [porque] el tiempo de ciclo del procesador disminuye a una tasa más rápida que el tiempo de ciclo de memoria... Esta tónica aumenta el número de ciclos involucrados en la ejecución de instrucciones, así como el tamaño relativo de la penalización por bifurcación.”

Michael J. Flynn, Arquitectura de computadoras: Diseño de procesador encauzado y paralelo, 1995

“Procedimiento para transitar por el pabellón de maternidad del HMO: 1) Levante la ventana de entrega, 2) Empuje, 3) Pague en la ventana del cajero, 4) Recoja al bebé. ¡Que tenga un buen día!”

Basado en una caricatura “Non Sequitur”, de Wiley

TEMAS DEL CAPÍTULO

- 15.1** Conceptos de *pipelining*
- 15.2** Atascos o burbujas encauzadas
- 15.3** Temporización y rendimiento encauzado
- 15.4** Diseño de rutas de datos encauzadas
- 15.5** Control encauzado
- 15.6** *Pipelining* óptimo

El encauzamiento (*pipelining*), alguna vez una técnica exótica reservada para las computadoras de primera línea, se convirtió en algo común en el diseño de computadoras. El encauzamiento en los procesadores se basa en el mismo principio de las líneas de ensamblaje en las fábricas: no hay necesidad de esperar hasta que una unidad esté completamente ensamblada antes de comenzar a trabajar con la siguiente. Sin embargo, existen algunas diferencias que forman el núcleo de las discusiones en éste y en próximos capítulos. Por ejemplo, mientras que un automóvil constituye una entidad completamente independiente de la que le precede y sigue en la línea de ensamblaje, no se puede decir lo mismo acerca de las instrucciones; existen dependencias de datos y control que se deben respetar para garantizar una operación correcta. En este capítulo se tienden los cimientos de la ejecución de instrucciones encauzada, se discute cómo manipular señales de control y se apuntan algunos de los retos. Las soluciones a los retos más difíciles se cubrirán en el siguiente capítulo.

■ 15.1 Conceptos de *pipelining*

Se han completado dos diseños para MicroMIPS; el diseño de ciclo sencillo del capítulo 13, con tasa de reloj de 125 MHz y CPI de 1, y el diseño multiciclo del capítulo 14, con tasa de reloj de 500 MHz

y CPI de casi 4. Ambos diseños ofrecen un rendimiento de aproximadamente 125 MIPS. ¿Hay alguna forma de superar éste? Existen al menos dos estrategias para lograr mayor rendimiento:

Estrategia 1: Usar múltiples rutas de datos independientes que puedan aceptar muchas instrucciones que se lean a la vez; esto último conduce a la organización de *emisión múltiple de instrucciones* o *superescalar*.

Estrategia 2: Traslapa la ejecución de varias instrucciones en el diseño de ciclo sencillo, comenzando la siguiente instrucción antes de que la previa haya concluido; lo anterior conduce a la organización encauzada (*pipelined*) o superencauzada (*superpipelined*).

La discusión de la primera estrategia se pospondrá para la sección 16.6, y en el resto de este capítulo se enfocará la atención en la segunda.

Considere el proceso de cinco etapas para registrarse al inicio de un curso en la universidad. Los estudiantes, después de llenar los formatos de selección de curso, deben proceder a través de: 1) buscar aprobación de un asesor académico, 2) pagar en la ventana del cajero, 3) devolver el formato de selección de curso y evidenciar el pago al registrar, 4) tomarse la fotografía para la identificación, 5) recoger esta última, recibo y calendario de clases. Suponga que cada una de tales etapas tarda casi dos minutos y se realiza en una ventilla ubicada en una gran sala. Se podría admitir un estudiante en la sala, dejarlo terminar el proceso de registro en alrededor de diez minutos y admitir a otro estudiante luego que el primero hubiese salido de la habitación. Esto último corresponde al diseño de ciclo sencillo del capítulo 13. Un mejor enfoque consiste en admitir un estudiante en la habitación cada dos minutos, porque, después de que el primer estudiante se hubiese movido de la etapa de aprobación a la etapa de pago, el asesor académico puede ayudar al siguiente estudiante, y así en el resto de la línea. Esta estrategia de línea de montaje, conocida como *pipelining* (encauzamiento), permite aumentar el procesamiento durante todo el desarrollo sin usar algún recurso adicional. La mejoría se logra al hacer uso completo de los recursos que permanecerían inactivos (*idle*) en el enfoque de ciclo sencillo.

Los cinco pasos en el proceso de registro de estudiantes, bosquejado en la figura 15.1, se pueden vincular con los cinco pasos de ejecución de instrucciones en MicroMIPS: 1) lectura (*fetch*) de instrucciones, 2) decodificación de instrucción y acceso de registro, 3) operación ALU, 4) acceso a memoria de datos y 5) escritura de registro. En virtud de que cada uno de estos pasos toma 1-2 ns, como se supuso en la sección 13.6, se puede iniciar la ejecución de una nueva instrucción cada 2 ns, siempre que el estado de cada instrucción y sus resultados parciales se salven en registros *pipeline* (tubería) ubicados entre etapas *pipeline*. La figura 15.2 muestra la ejecución de cinco instrucciones en tal forma (*pipelined*). La similitud de este enfoque con la línea de ensamblado industrial es obvia. Para el caso de la línea de montaje en una fábrica de automóviles, un auto puede tardar una hora en ensamblarse, pero, después de que el primer automóvil sale de la línea de montaje, los subsiguientes emergen minutos después uno detrás de otro.

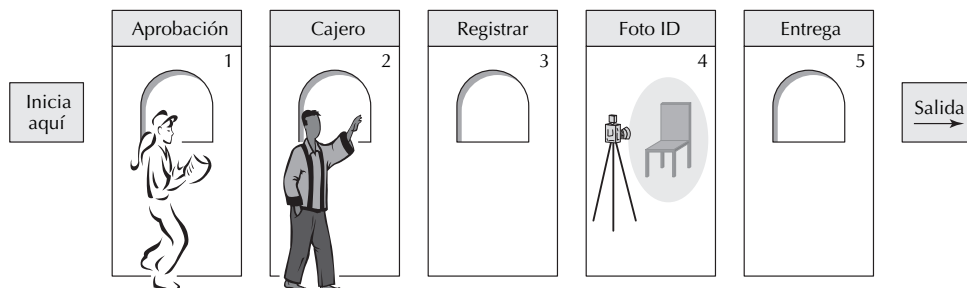


Figura 15.1 Pipelining en el proceso de registro de estudiantes.

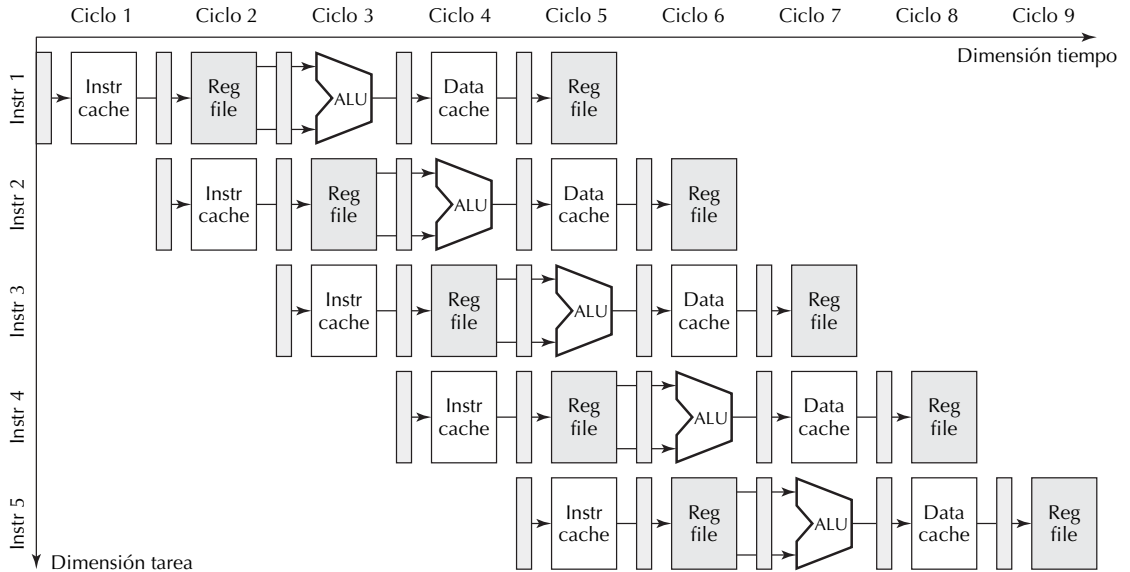


Figura 15.2 *Pipelining* en el proceso de ejecución de instrucciones MicroMIPS.

Como en toda analogía, la correspondencia entre la ejecución de instrucciones y el registro de estudiantes o el ensamblado de automóviles es imperfecto. Por ejemplo, mientras que los automóviles en una línea de ensamblado son independiente uno de otro, las instrucciones pueden y de hecho dependen del resultado de las instrucciones previas. Una instrucción que lee el contenido de un registro es totalmente dependiente de otra instrucción previa que carga un valor en el mismo registro. De igual modo, la ejecución de una instrucción que sigue una instrucción *branch* en la *pipeline* (tubería) depende de si la condición *branch* se satisface.

La representación gráfica de una *pipeline* en la figura 15.2 se conoce como diagrama tarea-tiempo (*task-time diagram*). En un diagrama tarea-tiempo, las etapas (*stages*) de cada tarea se alinean horizontalmente y sus posiciones a lo largo del eje horizontal representan la temporización (*timing*) de su ejecución. Una segunda representación gráfica de una *pipeline* es el diagrama espacio-tiempo que se muestra en la figura 15.3*b*, junto a una forma más abstracta del correspondiente diagrama tarea-tiempo en la figura 15.3*a*. En un diagrama espacio-tiempo, el eje vertical representa las etapas en la *pipeline* (la dimensión espacio) y los recuadros que representan las diversas etapas de una tarea se alinean diagonalmente.

Cuando un número finito de tareas se ejecuta en una *pipeline*, el diagrama espacio-tiempo claramente muestra la *región de arranque* de la *pipeline*, donde todas las etapas todavía no se utilizan por completo, y la *región de drenado* de la *pipeline*, compuesta de etapas que han quedado inactivas porque la última tarea las ha dejado. Si una *pipeline* ejecuta muchas tareas, se puede ignorar la cabecera (*overhead*) de arranque (*start-up*) y drenaje (*drainage*), y el rendimiento total efectivo de la *pipeline* toma una tarea por ciclo. En términos de ejecución de instrucciones, lo anterior corresponde a un CPI efectivo de 1. Si la *pipeline* de cinco etapas se tiene que drenar después de ejecutar siete instrucciones (figura 15.3*b*), entonces siete instrucciones se ejecutan en 11 ciclos, ello hace el CPI efectivo igual a $11/7 = 1.57$. Por tanto, ganar el beneficio total de una *pipeline* requiere que no se drene con demasiada frecuencia. Como se verá más tarde, esto es muy difícil de garantizar para la ejecución de instrucciones debido a las dependencias de dato y control.

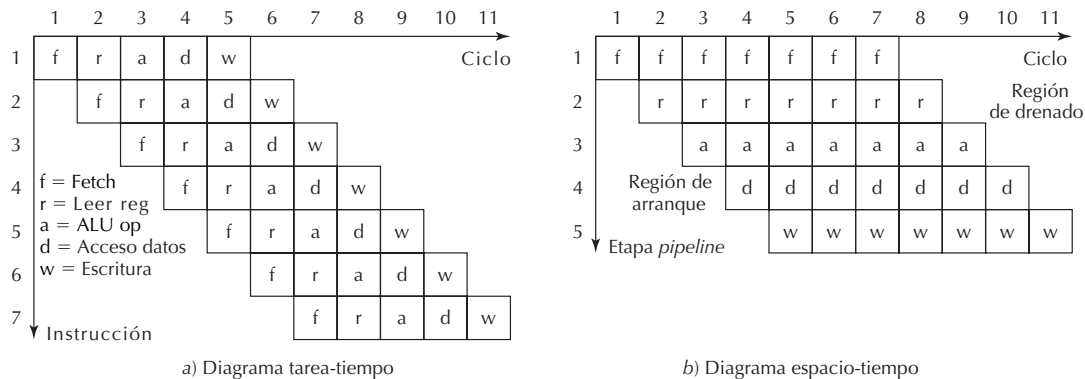


Figura 15.3 Dos representaciones gráficas abstractas de una *pipeline* de cinco etapas que ejecuta siete tareas (instrucciones).

De manera ideal, una *pipeline* de q etapas puede aumentar el rendimiento total de ejecución de instrucciones por un factor de q . Esto no es en mucho el caso, debido a:

1. Efectos de arranque y drenado de *pipeline*.
2. Desperdicios debido a retardos de etapa desiguales.
3. Cabecera de tiempo de salvar los resultados de etapa en registros.
4. Margen de seguridad en periodo de reloj necesario por sesgo de reloj.

El primer factor se discutirá más adelante en este capítulo y se ignorarán los dos últimos factores. El desperdicio debido a retardos de etapa desiguales es evidente a partir del *pipeline* MicroMIPS en el que los retardos de etapa son 2, 1, 2, 2 y 1 ns. Incluso ignorando todas las otras cabeceras, esto último conduce a una tasa de reloj de 500 MHz para la implementación encauzada de cinco etapas y un factor de aceleración de cuando mucho 4 en lugar del factor ideal de 5 relativo a la implementación de ciclo sencillo.

Ejemplo 15.1: Rendimiento total de una fotocopidora Una fotocopidora con un alimentador de documentos capaz de contener x hojas copia la primera hoja en 4 s y cada hoja subsecuente en 1 s. La ruta del papel de la copiadora se puede visualizar como una *pipeline* de cuatro etapas, donde cada etapa tiene una latencia de 1 s. La primera hoja pasa a través de las cuatro etapas *pipeline* y emerge en la salida después de 4 s. Cada hoja subsecuente también se copia en 4 s, pero emerge 1 s después de la hoja previa. Discuta cómo el rendimiento total de esta fotocopidora varía con x , si supone que cargar el alimentador de documentos y remover las copias tarda 15 s en conjunto. Además, compare el rendimiento total de la fotocopidora con el de una unidad no encauzada que copia cada hoja en 4 s.

Solución: Cargar el alimentador de documentos con hojas de entrada y descargar las copias de salida constituyen las partes de cabeceras arranque y drenado. Mientras más grande sea el valor x , más pequeño será el efecto de estas cabeceras en el rendimiento total. En este contexto, cuando x tiende al infinito se logra un rendimiento total estacionario de una copia por segundo, ello propicia una aceleración de 4 sobre una copiadora no encauzada. El rendimiento bajo condiciones más realistas se deriva fácilmente. Por ejemplo, con suposiciones específicas, cada lote de x hojas se copiará en $15 + 4 + (x - 1) = 18 + x$ segundos. Lo anterior se debe comparar con $4x$ segundos para una copiadora sin *pipelining*. Por tanto, para $x > 6$, se logra ganancia en rendimiento, y para $x = 50$, la aceleración es $200/68 = 2.94$.

15.2 Atascos o burbujas encauzadas

La dependencia de datos en *pipelines* (ejecución de una instrucción dependiendo de la conclusión de una instrucción previa) puede causar que aquélla se atasque (*stall*), lo cual disminuye el rendimiento. Existen dos tipos de dependencia de datos para la *pipeline* MicroMIPS:

Leer después de calcular: acceso a registro después de actualizarlo con un valor calculado.

Leer después de cargar: acceso a registro después de actualizarlo con datos de la memoria.

En la figura 15.4 aparecen dos ejemplos de dependencia de datos leer después de —calcular, donde la tercera instrucción usa el valor que la segunda instrucción escribe en el registro \$8 y la cuarta instrucción necesita el resultado de la tercera instrucción en el registro \$9. Observe que escribir en el registro \$8 se completa en el ciclo 6; por tanto, la lectura del nuevo valor del registro \$8 es posible al comenzar en el ciclo 7. Sin embargo, la tercera instrucción lee los registros \$8 y \$2 en el ciclo 4; por tanto, no obtendrá el valor pretendido para el registro \$8. Este problema de dependencia de datos se puede resolver mediante *inserción de burbujas* (*bubble insertion*) o mediante *adelantamiento de datos* (*data forwarding*).

Inserción de burbuja

La primera solución es que el ensamblador detecte este tipo de dependencia de datos e inserte tres instrucciones redundantes, pero inocuas (agregar 0 a un registro o correr un registro por 0 bits), antes de la siguiente instrucción. Tal instrucción, ejemplificada por la instrucción todos 0 en MicroMIPS, a veces se denomina *no-op* (abreviatura para *no operación*). Puesto que no realizan trabajo útil y sólo ocupan localidades de memoria para espaciar las instrucciones dependientes de datos, tales instrucciones *no-op* tienen el papel de *burbujas* en una tubería de agua. Se dice que el ensamblador inserta tres burbujas en la *pipeline* para resolver una dependencia de datos leer después de calcular. En realidad, insertar dos burbujas puede ser suficiente si se observa que escribir en y leer de un registro toman cada uno 1 ns; de modo que es factible diseñar el archivo de registro para hacer que el valor escrito en un registro en la primera mitad de un ciclo de 2 ns esté disponible en la salida en la segunda mitad del mismo ciclo.

La inserción de burbujas en una *pipeline* implica rendimiento total reducido; obviamente, la inserción de tres burbujas daña más el rendimiento que insertar dos burbujas. Por tanto, un objetivo de los

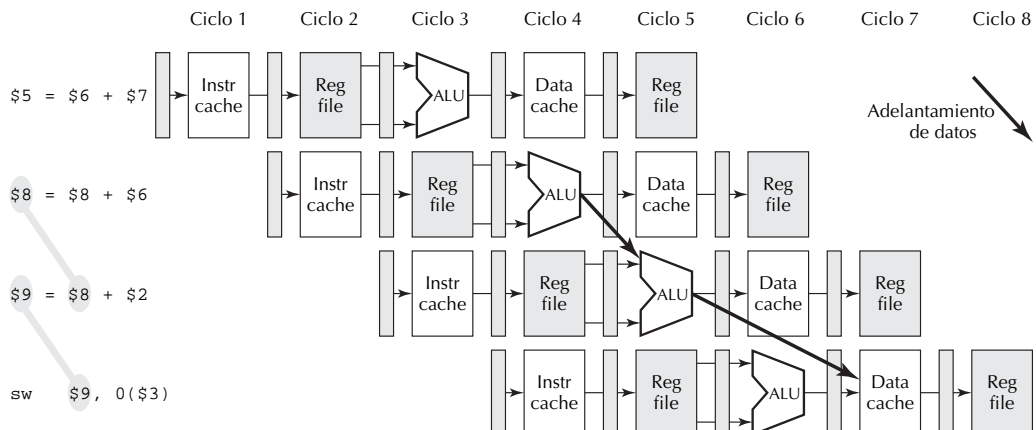


Figura 15.4 Dependencia de datos leer-después-de-calcular, y su posible resolución a través del adelantamiento de datos.

diseñadores de *pipeline* consiste en minimizar las instancias de inserción de burbuja. Un compilador inteligente también puede ayudar a mitigar la penalización por rendimiento. Si las instrucciones i y $i + 1$ tienen dependencia de datos leer-después-de-calcular, el compilador puede observar antes de la instrucción i o después de la instrucción $i + 1$ en el programa para ver si existe alguna instrucción que se puede reubicar entre las dos instrucciones dependientes de datos sin cambiar el funcionamiento correcto del programa. Si, por ejemplo, se encuentran dos de estas instrucciones y se mueven, sólo será necesario insertar una burbuja, ello conduce a mejoría en el rendimiento.

Adelantamiento de datos

Observe que en la figura 15.4, aun cuando el resultado de la segunda instrucción todavía no se almacena en el registro \$8 para cuando se necesita la tercera instrucción, el resultado está disponible en la salida de la ALU. En consecuencia, al proporcionarse una ruta desviada desde la salida de la ALU hacia una de sus entradas, el valor necesario se puede trasladar desde la segunda hasta la tercera instrucción y usarse por la última, aun cuando el registro \$8 todavía no contenga el valor correcto. Este enfoque se denomina *adelantamiento de datos*. En la sección 16.2 se proporcionarán detalles de implementación para el adelantamiento de datos.

Ejemplo 15.2: Inserción de burbuja En la figura 15.4, ¿cuántas burbujas se deben insertar para la segunda dependencia de datos leer-después-de-calcular y que involucra el registro \$9? ¿Y si la última instrucción de la figura 15.4 se cambia por `sw $3, 0($9)`?

Solución: Todavía se necesitan tres burbujas, pues la lectura de registro se realiza en dos ciclos antes de la correspondiente escritura, mientras que ésta se debe realizar un ciclo después. Al igual que antes, si la lectura y escritura de registro pueden ocurrir en el mismo ciclo de 2 ns, dos burbujas serán suficientes. Desde luego, el adelantamiento de datos puede obviar la necesidad de burbujas (figura 15.4). Si la última instrucción se sustituye por `sw $3, 0($9)`, sería necesario el contenido del registro \$9 como entrada a la ALU para calcular la dirección. Esto último todavía involucraría la inserción de tres (o dos) burbujas.

En la figura 15.5 se muestra una dependencia de datos leer-después-de-cargar, donde la tercera instrucción usa el valor que la segunda instrucción carga en el registro \$8. Sin adelantamiento de datos, la tercera instrucción debe leer el registro \$8 en el ciclo 7, un ciclo después de que la segunda instrucción completa el proceso de carga. Lo anterior implica la inserción de tres burbujas en la *pipeline* para evitar el uso del valor equivocado en la tercera instrucción. Observe que, para este caso, el adelantamiento de datos solo no puede resolver el problema. El valor necesario por la ALU en la tercera instrucción se vuelve disponible al final del ciclo 5; de modo que, incluso si está disponible un mecanismo de adelantamiento de datos (que se discute en la sección 16.2), todavía se necesita una burbuja para asegurar la operación correcta. Un compilador inteligente puede buscar antes la instrucción `lw` para ver si existe alguna instrucción que se pueda colocar después de `lw` sin afectar la semántica del programa. Si es así, la rendija de burbuja no se desperdiciaría. Este tipo de reordenamiento de instrucción es común en los compiladores comunes.

Otra forma de dependencia entre instrucciones la constituye la *dependencia de control*. Esto último se ejemplifica mediante una instrucción *branch* condicional. Cuando se ejecuta un *branch* condicional, la ubicación de la siguiente instrucción depende de si esta condición se satisface. Sin embargo, como consecuencia de que las instrucciones *branch* en MicroMIPS se basan en probar los contenidos de los registros (signo de uno o comparación de dos), lo más pronto que se puede esperar resolver la condición *branch* es al final de la segunda etapa *pipeline*. En consecuencia, se requiere una burbuja después

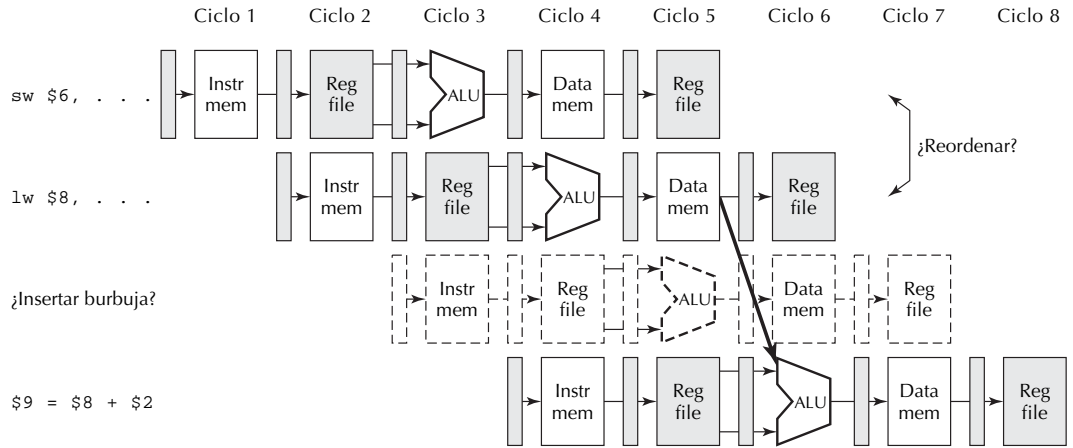


Figura 15.5 Dependencia de datos leer-después-de-cargar y su posible resolución a través de inserción de burbuja y adelantamiento de datos.

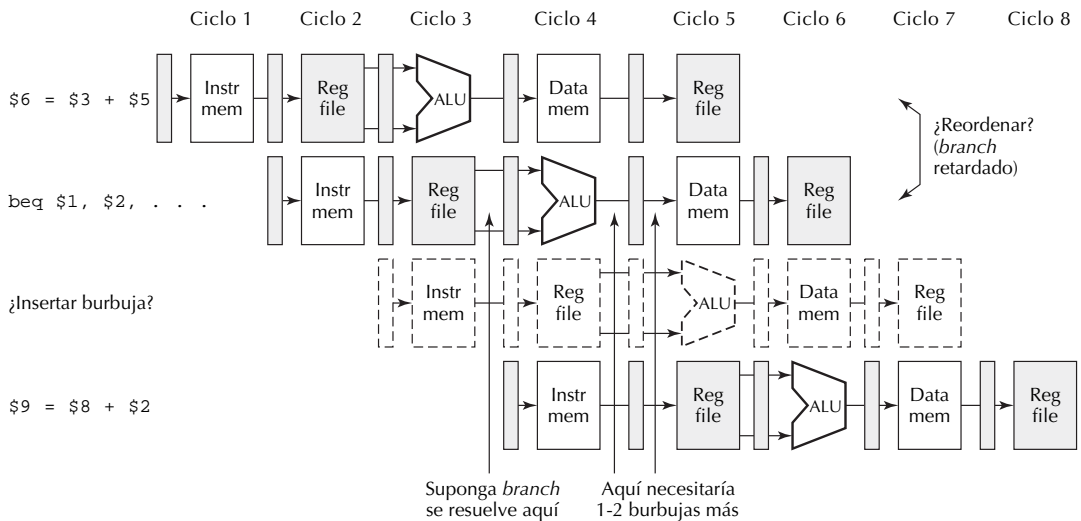


Figura 15.6 Dependencia de control debida a *branch* condicional.

de cada instrucción *branch* condicional. Ello puede ser cierto incluso para una instrucción *jump* incondicional si la decodificación de la instrucción comienza en el segundo ciclo, lo que evita aprender que la instrucción represente un salto a tiempo para modificar el PC para la siguiente instrucción leída (*fetch*). Para el caso de dependencias de datos, observe que el adelantamiento de datos obvia la necesidad de más burbujas que de otro modo serían necesarias. ¿Existe algún método correspondiente para evitar burbujas después de las instrucciones *branch*?

Un posible método es usar bifurcaciones retardadas. En otras palabras, la definición de una instrucción *branch* se modifica de modo que *beq. . .* significa “verifica la condición, ejecuta la siguiente instrucción en secuencia, luego bifurca a la dirección indicada si la condición *branch* se satisface”. Se dice que la instrucción adicional que se ejecuta antes de que ocurra la bifurcación ocupa la *rendija de retardo de bifurcación*. En el ejemplo de la figura 15.6, el reordenamiento de las primeras dos ins-

trucciones y la consideración de la bifurcación como de la variedad retardada obvian la necesidad de insertar una burbuja.

Observe que verificar la condición *branch* en la etapa 2 de la *pipeline* requiere la provisión de un comparador para señalar la igualdad o desigualdad de los contenidos de registro para el caso de las instrucciones *beq* o *bne*, respectivamente. Se podría evitar este circuito adicional, y en vez de ello optar por el uso de la ALU para realizar una resta y luego deducir la igualdad o desigualdad con base en la bandera de salida *zero* de la ALU. Sin embargo, esto último aumentaría la penalización de *branch* a dos o tres burbujas, dependiendo de si, después de la operación ALU, todavía queda suficiente tiempo disponible en la etapa 3 de la *pipeline* para establecer el PC a la dirección blanco de bifurcación. Este es un precio muy alto por evitar un circuito comparador simple.

■ 15.3 Temporización y rendimiento encauzado

Como ya se afirmó, una *pipeline* de q etapas aumenta idealmente el rendimiento total de ejecución de instrucción por un factor de q . Esto sucede porque sin alguna cabecera o dependencia de cualquier tipo, y suponiendo que una tarea de latencia t es divisible en q etapas que tienen iguales latencias t/q , una nueva tarea se puede iniciar cada t/q unidades de tiempo en lugar de cada t unidades de tiempo sin *pipelining*. En esta sección la meta es determinar el rendimiento real ganado al modelar los efectos de las cabeceras y dependencias.

La cabecera debida a latencias de etapa desiguales y *latching* de señales entre etapas se puede modelar al tomar el retardo de etapa como $t/q + \tau$ en vez del ideal t/q , como en la figura 15.7. En este sentido, el aumento en el rendimiento total caerá del ideal de q a

$$\text{Aumento de rendimiento total en una } pipeline \text{ de } q \text{ etapas} = \frac{t}{t/q + \tau} = \frac{q}{1 + q\tau/t}$$

De modo que se puede acercar el factor de mejora de rendimiento total ideal de q , siempre que la cabecera $q\tau$ de *latching* de *pipeline* acumulada sea más pequeña que t . La figura 15.8 grafica la mejora en rendimiento total como función de q para diferentes razones de cabecera τ/t .

La ecuación de rendimiento total de *pipeline* y la figura 15.8 son reminiscencias de la ley de Amdahl (sección 4.3). De hecho, la fórmula de aceleración de Amdahl se puede escribir así para exponer aún más las similitudes:

$$\text{Aceleración con mejora } p \text{ en la fracción } 1 - f = \frac{1}{f + (1 - f)/p} = \frac{p}{1 + (p - 1)f}$$

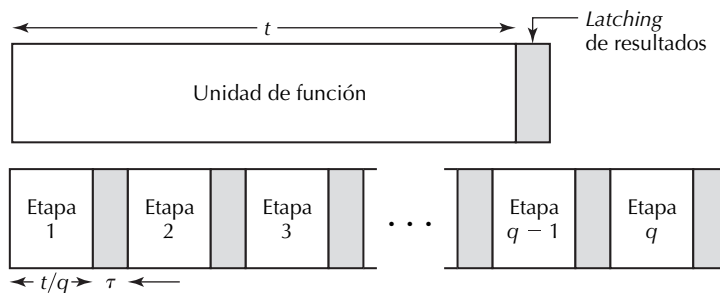


Figura 15.7 Forma encauzada de una unidad de función con cabecera *latching*.

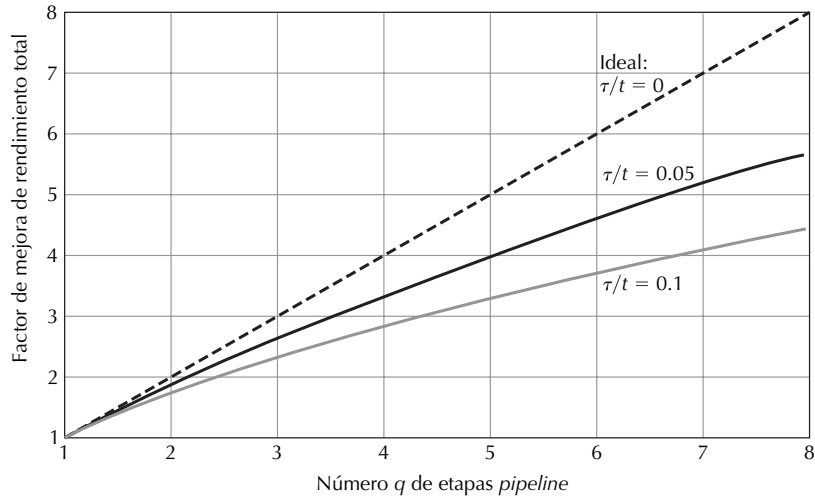


Figura 15.8 Mejora de rendimiento total debida a *pipelining* como función del número de etapas *pipeline* para diferentes cabeceras de esta clase.

Observe que τ/t (magnitud relativa del tiempo por etapa de la cabecera τ) es tan importante como la fracción f no afectada o no mejorada de Amdahl; ambas limitan el rendimiento máximo que se puede lograr.

Para la implementación MicroMIPS encauzada (*pipelined*), se tiene $t = 8$ ns (que corresponde a la latencia en implementación de ciclo sencillo), $q = 5$ y $q\tau = 2$ ns. Esto último conduce a un factor de mejora de rendimiento total de 4 con la *pipeline* de cinco etapas. Por ende, el rendimiento de la implementación encauzada es de 500 MIPS frente a 125 MIPS para el diseño de ciclo sencillo. Sin embargo, observe que todavía no se consideran los efectos de las dependencias de datos y control.

A continuación se intenta modelar los efectos de las dependencias de datos y control. Con el propósito de mantener el análisis simple, se supone que la mayoría de las dependencias de datos se resuelven mediante adelantamiento de datos, con una sola burbuja insertada en la *pipeline* sólo para una dependencia de datos leer-después-de-cargar (figura 15.5). De igual modo, una fracción de las instrucciones *branch* (para las que el compilador no tiene éxito en llenar la rendija de retardo de bifurcación con una instrucción útil) conduce a la inserción de una sola burbuja en la *pipeline*. En cualquier caso no resulta ningún tipo de dependencia en la inserción de más de una burbuja. Sea β la fracción de todas las instrucciones que se siguen por una burbuja en la *pipeline*. Tales burbujas reducen el rendimiento total por un factor de $1 + \beta$, ello propicia la ecuación de rendimiento total

$$\text{Aumento en el rendimiento total con dependencias} = \frac{q}{(1 + q\tau/t)(1 + \beta)}$$

Considere ahora la mezcla de instrucciones que se usó antes en la evaluación de los diseños de ciclo sencillo y multiciclo:

Tipo R	44%
Load	24%
Store	12%
Branch	18%
Jump	2%

Es razonable suponer que casi a una cuarta parte de la bifurcación y la instrucción *load* le seguirán burbujas en la *pipeline*. Esto último corresponde a $0.25(0.24 + 0.18) = 0.105 = \beta$ o 10.5% de todas las instrucciones. Con lo anterior y las suposiciones previas, el rendimiento de la implementación MicroMIPS encauzada se estima en:

$$\frac{500 \text{ MIPS}}{1.105} = 452 \text{ MIPS}$$

Incluso si se procede con la suposición más pesimista de que a la mitad, más que a la cuarta parte de las instrucciones *load* y *branch* seguirán burbujas en la *pipeline*, el rendimiento todavía sería $500/1.21 = 413$ MIPS, que es más de 3.3 veces el del diseño de ciclo sencillo o multiciclo.

Ejemplo 15.3: CPI efectivo Calcule el CPI efectivo para la implementación MicroMIPS encauzada con las suposiciones de cabecera dadas en la discusión anterior.

Solución: Excepto por el efecto de las burbujas, una nueva instrucción inicia en cada ciclo de reloj. En virtud de que una fracción β de las instrucciones se sigue por una sola burbuja en la *pipeline*, el CPI efectivo es $1 + \beta$. Ello conduce a un CPI de 1.105 o 1.21 para $\beta = 0.105$ o 0.21, respectivamente. Con base en este resultado, se observa que la implementación encauzada combina los beneficios de las implementaciones de ciclo sencillo y multiciclo en que casi logra el CPI bajo del anterior con la mayor frecuencia de reloj del último.

La discusión hasta ahora se basó en la suposición de que una función con una latencia de t se puede dividir fácilmente en q etapas de igual latencia t/q . En la práctica, con frecuencia éste no es el caso. Por ejemplo, la ruta de datos de MicroMIPS de ciclo sencillo tiene una latencia de 8 ns, que se divide en cinco etapas, cada una con una latencia de 2 ns (no $8/5 = 1.6$ ns). El efecto de esta función de subdivisión de cabecera sobre el rendimiento es el mismo que el de la cabecera *latching* τ . En otras palabras, el aumento en retardo de etapa del ideal de 1.6 ns a 2 ns se puede ver como correspondiente a una cabecera de etapa de $\tau = 0.4$ ns. En consecuencia, el parámetro τ se usa para incorporar tanto la cabecera *latching* y la función subdivisión de cabecera.

■ 15.4 Diseño de rutas de datos encauzadas

La ruta de datos encauzada para MicroMIPS se obtiene mediante la inserción de *latches* o registros en la ruta de datos de ciclo sencillo de la figura 13.3. El resultado se muestra en la figura 15.9. Las cinco etapas *pipeline* son: 1) *fetch* instrucción, 2) acceso a registro y decodificación de instrucción, 3) operación ALU, 4) acceso a caché de datos y 5) escritura de registro.

La comparación de la ruta de datos encauzada de la figura 15.9 con la versión no encauzada de la figura 13.3 revela algunos cambios y adiciones además de *latches* interetapas:

- Inclusión de un incrementador y un mux dentro de la lógica del estado 1.
- Cambio en colocación para el mux que originalmente alimenta el archivo de registro.
- Cambio en el número de entradas para el mux que alimenta la ALU.
- División del mux de la extrema derecha en dos muxes en las etapas 4 y 5.

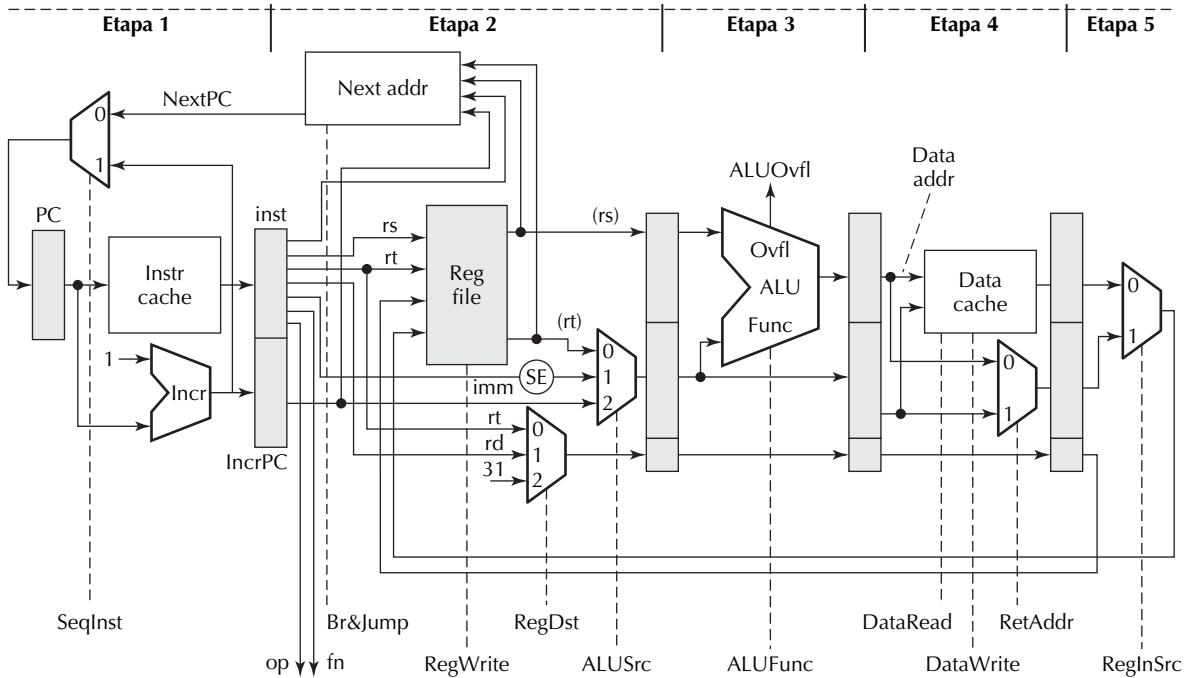


Figura 15.9 Elementos clave de la ruta de datos MicroMIPS encauzada.

Las razones para estas diferencias serán claras conforme la función de las cinco etapas de *pipeline* se describa en los siguientes párrafos.

Etapa *pipeline* 1 (instrucción *fetch*): Esta etapa es muy similar al segmento de la instrucción *fetch* de la ruta de datos de ciclo sencillo en la figura 13.3. Una diferencia clave es que se incluyó un incrementador para calcular $(PC) + 4$, al agregar 1 a los 30 bits superiores en el contador de programa. En la ruta de datos de ciclo sencillo, este incremento se combinó con la lógica de dirección de bifurcación. Aquí, dado que la decisión de bifurcación y el cálculo de dirección se realizan en la segunda etapa de *pipeline*, se necesita aumentar el PC en la etapa 1 para tener la capacidad de *fetch* la instrucción siguiente. Este valor PC aumentado se salva en un registro especial entre las etapas 1 y 2 para usar en la etapa 2. La señal de control SeqInst se despostula a menos que la instrucción actualmente en la segunda etapa de *pipeline* sea una *branch* o *jump*, en cuyo caso la salida NextPC del recuadro Next addr en lo alto de la figura 15.9 se debe escribir en el PC.

Etapa *pipeline* 2 (leer registro y decodificación de instrucción): Al comparar esta etapa con el segmento correspondiente de la ruta de datos de ciclo sencillo en la figura 13.3, se notan algunos cambios. Primero, el multiplexor que elige el registro al que se escribirá no está directamente conectado con el archivo de registro sino más bien a un registro especial entre las etapas 2 y 3. Esto último sucede porque la acción de escribir en el registro elegido ocurre no en esta etapa sino más bien en la 5. Por esta razón, el índice del registro al que se escribirá se lleva con la instrucción conforme se mueve entre etapas y eventualmente se usa en la etapa 5 para la operación de escritura. El multiplexor superior en esta segunda etapa permite portar (rt) a la siguiente etapa o el operando inme-

diato extendido en signo o el valor PC aumentado. Las opciones 0 y 1 corresponden a la función del multiplexor ubicado entre el archivo de registro y la ALU en la figura 13.3. La opción 2 se necesita sólo para las instrucciones `jal` y `syscall` que requieren que el valor PC aumentado se escriba en el registro `$31`.

Etapla pipeline 3 (operación ALU): Esta etapa es muy directa. Se instruye a la ALU, a través de las señales de control `ALUFunc`, para que realice una operación en sus dos entradas. A diferencia de la ruta de datos de ciclo sencillo de la figura 13.3, no se involucra selección para la entrada inferior de la ALU, pues ésta se realizó en la etapa 2 antes de salvar la entrada elegida.

Etapla pipeline 4 (acceso a memoria de datos): Esta etapa es muy similar a su segmento correspondiente en la ruta de datos de ciclo sencillo de la figura 13.3. Una diferencia clave es la adición de un multiplexor para permitir que el valor PC aumentado se envíe a la etapa 5, donde se escribe en el registro `$31`. Esto sólo se necesita para las instrucciones `jal` y `syscall`.

Etapla pipeline 5 (escritura de registro): En esta etapa, los datos se escriben en un registro si lo requiere la instrucción que se ejecuta. La necesidad de escritura se indica a través de la postulación de la señal de control `RegWrite`. El registro donde se escribirá se seleccionó previamente en la etapa 2 de la *pipeline*, y su índice pasó entre las etapas en un campo de cinco bits (la parte inferior de los registros *pipeline* entre las etapas 2-3, 3-4 y 4-5 de la figura 15.9). Los datos a escribir son la salida de la caché de datos o uno de los dos valores elegidos en la etapa 4 y se salvan en un registro entre las etapas 4-5.

El bloque `Next addr` de la figura 15.9 es similar al bloque correspondiente en la ruta de datos de ciclo sencillo de la figura 13.3 representado en la (figura 13.4), la única diferencia es que la entrada $(PC)_{31:2}$ al circuito está ahora en una forma aumentada debido al incrementador de la etapa 1. Este cambio hace que la entrada de acarreo del sumador en la figura 13.4 se fije en 0 en lugar de 1, y la salida superior del circuito no se use.

Ejemplo 15.4: Contenidos de registros *pipeline* La siguiente secuencia de instrucciones Micro-MIPS se almacena en memoria, comenzando en la dirección hexa `0x00605040`:

```
lw    $t0, 4($t1)    # $t0 ← mem[4+($t1)]
add   $t2, $s2, $s3   # $t2 ← ($s2) + ($s3)
xor    $t3, $s4, $s5   # $t3 ← ($s4) ⊕ ($s5)
```

Determine los contenidos sucesivos de los registros *pipeline* de la figura 15.9 conforme estas instrucciones se ejecutan, comenzando con una *pipeline* vacía. Suponga que el registro `$si` retiene el entero `4i`, el registro `$t1` retiene `0x10000000` y el contenido de la localidad `0x10000004` en memoria es `0x10203040`.

Solución: La primera instrucción carga la palabra `0x10203040` en el registro `$8` (vea la figura 5.2 para las convenciones de denominación de registro). La segunda instrucción coloca la suma $0x00000008 + 0x0000000c = 0x00000014$ en el registro `$10`. La tercera instrucción coloca $0x00000010 \oplus 0x00000014 = 0x00000004$ en el registro `$11`. La tabla 15.1 muestra los contenidos del registro *pipeline* en el curso de ejecución de estas tres instrucciones. Las entradas en blanco de la tabla que siguen a estas últimas representan contenidos de registro desconocidos correspondientes a instrucciones posteriores. Los encabezados de columna representan los nombres de registro y las etapas entre las que aparecen los registros.

■ **TABLA 15.1** Contenidos de registros *pipeline* para el ejemplo 15.4.

Ciclo de reloj	PC(0-1)	Inst(1-2) IncrPC(1-2)	Top(2-3) Bottom(2-3) Reg(2-3)	Top(3-4) Bottom(3-4) Reg(3-4)	Top(4-5) Bottom(4-5) Reg(4-5)
1	0x00605040				
2	0x00605044	lw \$t0,4(\$t1) 0x00605044			
3	0x00605048	add \$t2,\$s2,\$s3 0x00605048	0x10000000 0x00000004 \$8		
4		xor \$t3,\$s4,\$s5 0x0060504c	0x00000008 0x0000000c \$10	0x10000004 0x10000000 \$8	
5			0x00000010 0x00000014 \$11	0x00000014 0x0000000c \$10	0x10203040 0x10000004 \$8
6				0x00000004 0x00000014 \$11	0x00000000 0x00000014 \$10
7					0x00000000 0x00000004 \$11

■ 15.5 Control encauzado

Conforme una instrucción se mueve mediante una *pipeline*, debe llevar consigo suficiente información de control para permitir la determinación de todas las acciones subsecuentes necesarias para completar su ejecución. Con referencia a la figura 15.9, cuando la instrucción se decodifica en la etapa 2, el decodificador de instrucción postula las siguientes señales de control:

Stages 1 and 2: SeqInst, Br&Jump, RegDst, ALUSrc

Las señales de control restantes se necesitarán en etapas (*stages*) futuras; por tanto, se deben salvar y enviar junto con los datos en los registros *pipeline* interetapas. De este modo, éstos deben tener espacio para almacenar las siguientes señales de control generadas en la etapa 2:

Interetapas 2-3: ALUFunc, DataRead, DataWrite, RetAddr, RegInSrc, RegWrite

Interetapas 3-4: DataRead, DataWrite, RetAddr, RegInSrc, RegWrite

Interetapas 4-5: RegInSrc, RegWrite

Por ende, las señales de control para las etapas 1 y 2 se generan en la etapa 2 y se usan enseguida, mientras que las señales de control para las etapas 3-5 se generan en la etapa 2 y se transmiten, junto con cualquier dato requerido para la instrucción, a etapas posteriores. La figura 15.10 muestra las extensiones requeridas de los registros del tipo *pipeline* interetapas para retener la información de control necesaria al final de las etapas (*stages*) 2, 3 y 4.

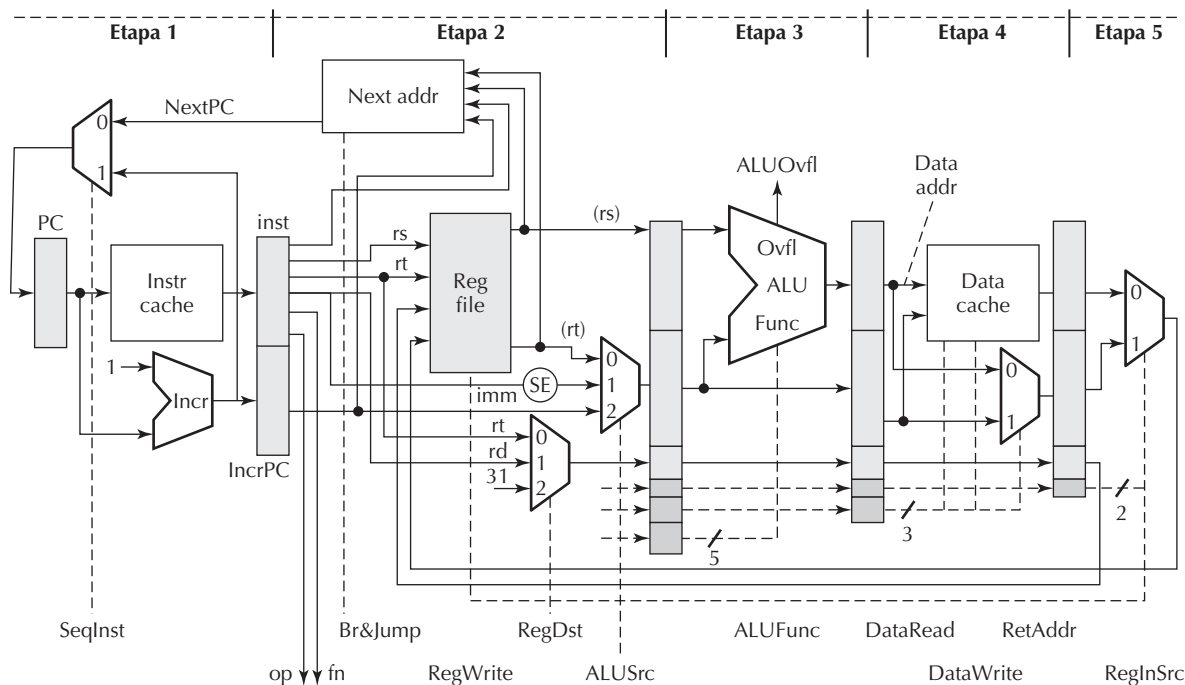


Figura 15.10 Señales de control encauzadas.

Ejemplo 15.5: Especificación de señales de control Para la secuencia de instrucciones dada en el ejemplo 15.4, especifique las señales de control que se retienen en los registros *pipeline* extendidos de la figura 15.10 conforme se ejecutan las tres instrucciones.

Solución: Con referencia a la tabla 14.3, se obtiene el código de función ALU de cinco bits en la etapa 3 (Add/Sub, LogicFn, FnClass), los tres bits de control para la etapa 4 (DataRead, DataWrite, RetAddr), y los dos bits de control para la etapa 5 (regInSrc, RegWrite). Los resultados se resumen en la tabla 15.2, donde los ciclos de reloj son los mismos que para la tabla 15.1. Los encabezados de columna consisten de la unidad o función que se controlará y las etapas entre las que se almacenan las señales de control.

TABLA 15.2 Valores de señal de control encauzados para el ejemplo 15.5.

Ciclo	ALU(2–3), Cache(2–3), Writeback(2–3)	Cache(3–4), Writeback(3–4)	Writeback(4–5)
3	0xx10, 10x, 01		
4	0xx10, 000, 11	10x, 01	
5	x1011, 000, 11	000, 11	01
6		000, 11	11
7			11

■ 15.6 Pipelining óptimo

Ahora se sabe que el rendimiento de un procesador se puede mejorar mediante la ejecución de instrucciones en una ruta de datos encauzada. Una *pipeline* que sólo tenga algunas etapas constituye una *shallow pipeline* (tubería superficial), mientras que una *pipeline* con muchas etapas representa una *deep pipeline* (tubería profunda). Desde luego, superficialidad y profundidad son términos relativos, de modo que la frontera entre las dos clases de tuberías es más bien confusa (como la diferencia entre personas bajas y altas). Como regla, una *pipeline* con media docena o menos de etapas se considera superficial, mientras que otra *pipeline* que tiene una docena o más de etapas se considera profunda. Aumentar el número σ de etapas de *pipeline*, o la *profundidad del pipelining*, conduce a mejora en el rendimiento. Sin embargo, éste no aumenta linealmente con σ . Ya se han discutido algunas razones para este escalamiento sublineal del rendimiento.

1. Las incertidumbres de cronometrado y los requisitos de establecimiento de *latches* imponen una cabecera de tiempo τ por etapa. Como consecuencia de esta cabecera, el rendimiento total de la *pipeline* mejora con subdivisiones más finas de la ruta de datos, pero el impacto de τ será más pronunciado conforme σ aumenta; incluso en el extremo de etapas *pipeline* con latencia despreciable, el rendimiento total nunca puede superar $1/\tau$.
2. Si las σ etapas de una *pipeline* no tienen exactamente la misma latencia, que usualmente es el caso debido a la irregularidad de las fronteras naturales en las que se pueden insertar *latches*, entonces, incluso sin cabecera, el rendimiento total mejorará por un factor menor que σ .
3. Las dependencias de datos y los riesgos *branch/jump* a veces hacen imposible mantener llenas las tuberías en todo momento. El reordenamiento de instrucciones por compiladores puede reducir la penalización de rendimiento de tales dependencias y riesgos, pero siempre hay situaciones en las que no se puede evitar totalmente la pérdida de rendimiento total.

En la implementación encauzada de MicroMIPS, se tienen las siguientes latencias de etapa:

- 2 ns Acceso a memoria de instrucción
- 1 ns Lectura y decodificación de registro
- 2 ns Operación ALU
- 2 ns Acceso a memoria de datos
- 1 ns Escritura de registro

Se ve que una pipeline de cinco etapas mejora el rendimiento total de la ejecución de instrucciones MicroMIPS mediante un factor 4 en el mejor de los casos (no hay cabecera de tiempo de etapa de ningún tipo ni atascos).

¿Existe alguna forma de lograr una mejora de rendimiento total superior a cuatro para MicroMIPS encauzado? Como primer paso, observe que, al menos teóricamente, el rendimiento se puede aumentar por un factor de 8 si se construye una *pipeline* con etapas de 1 ns. Esto último requeriría que el acceso de memoria y las operaciones ALU de 2 ns se dispersaran sobre dos etapas de *pipeline*. Para la ALU, las dos etapas requeridas se forman fácilmente con la inserción de *latches* a la mitad del camino en la ruta crítica del circuito lógico. Con referencia a la analogía dada en la figura 15.1, esto es como dividir la función de la ventana registrar en dos subfunciones, pues toma el doble de tiempo que cualquiera de las otras cuatro etapas. Dispersar la latencia de acceso a memoria a través de dos etapas de *pipeline* no es claro. Entender cómo se puede hacer lo anterior requiere conocimiento de cómo opera la memoria. Así que aquí se acepta que es factible una memoria encauzada de dos etapas y se difiere la discusión de la implementación a la sección 17.4.

La figura 15.11 contiene una representación abstracta de la *pipeline* resultante de ocho etapas para MicroMIPS y muestra cómo se ejecutan en ella instrucciones consecutivas. De nuevo, se supuso que

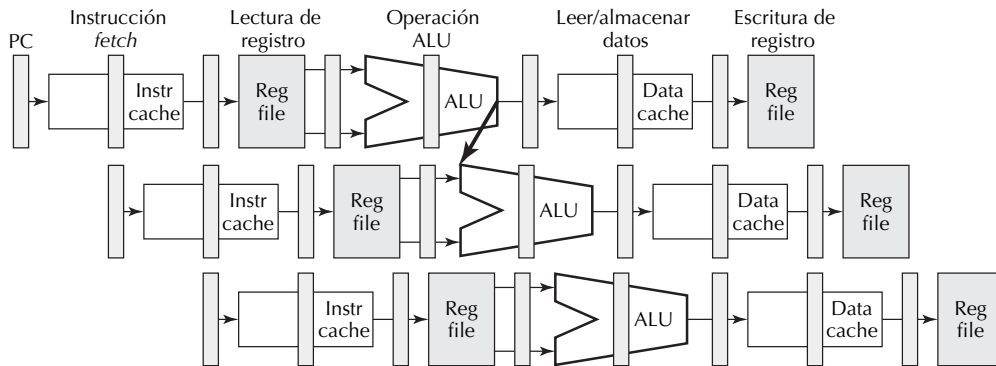


Figura 15.11 Ruta de datos encauzada de mayor rendimiento total para MicroMIPS y la ejecución de instrucciones consecutivas en ella.

la cabecera *pipelining* es despreciable; por tanto, cuando una operación de 2 ns se divide en dos etapas de *pipeline*, cada etapa tendrá una latencia de 1 ns.

Observe que con la *pipeline* más profunda de la figura 15.11, la condición *branch* se decide no más temprano que el final de la tercera etapa, en oposición a la segunda etapa de la figura 15.2. Ello implica la inserción de dos burbujas, o la provisión de dos rendijas de retardo, después de cada instrucción *branch*. Mientras que la experiencia muestra que una sola rendija de retardo *branch* casi siempre puede llenarse por un compilador, la segunda de las dos rendijas de retardo es un poco más difícil de usar y puede conducir a la inserción de una burbuja en la *pipeline*. Observe que el análisis precedente se basa en la suposición optimista de que el tiempo de ciclo de 1 ns todavía deja suficiente espacio para comparar contenidos de dos registros en el mismo ciclo cuando se leen. Esto último puede no ser factible, pero conduce a una burbuja o rendija de retardo adicionales. La situación para las instrucciones *jump* es similar en que también necesitan al menos dos burbujas. Sin embargo, éste es un problema menos serio, pues *jump* es menos frecuente que *branch*.

Las dependencias de datos también tienen consecuencias más serias en una *pipeline* más profunda. Considere, por ejemplo, una lectura de registro después de una instrucción previa que escribe en el mismo registro después de una operación ALU. Las flechas gruesas de la figura 15.11 muestran que, incluso si los datos se adelantan directamente de una instrucción a la siguiente, la dependencia precedente no se puede resolver sin la inserción de al menos una burbuja. Leer un registro después de una instrucción *load* precedente es más problemático y requiere tres burbujas (¿por qué?).

Ejemplo 15.6: Comparación de implementaciones encauzadas Compare el rendimiento de la implementación encauzada de ocho etapas de MicroMIPS en la figura 15.11, con la implementación de cinco etapas discutida antes aquí. Suponga que el compilador puede eliminar todas las dependencias de datos mediante un adecuado cronograma de instrucciones. Ignore las instrucciones *jump* y suponga que 20% de las instrucciones son *branch* condicionales para las que existen tres rendijas de retardo con la *pipeline* de ocho etapas: el compilador siempre puede llenar la primera rendija de retardo con una instrucción útil, pero, de las dos rendijas de retardo restantes, en promedio 1.5 quedan sin utilizar.

Solución: La *pipeline* de cinco etapas ejecuta instrucciones sin desperdicio debido a burbujas. Ello conduce a que se ejecuten 100 instrucciones en 200 ns, para un rendimiento de 500 MIPS. Las *pipeline* de ocho etapas desperdician 1.5 ciclos para 20 de cada 100 instrucciones. De modo que ejecuta 100

instrucciones en 130 ns, para un rendimiento de $1000 / 1.3 \cong 770$ MIPS. Observe que la razón $770/500 = 1.54$ de rendimientos totales es muy cercana a la razón $8/5 = 1.6$ del número de etapas *pipeline* en los dos diseños. Esto sólo es una indicación de las burbujas de bifurcación para la *pipeline* de ocho etapas más que la anulación de las ganancias de una reducción en el tiempo asignado para las operaciones de lectura y escritura de registro de 2 ns (un ciclo de reloj en la *pipeline* de cinco etapas) a 1 ns. Más reveladora es una comparación del rendimiento total de 770 MIPS de la *pipeline* de ocho etapas con el rendimiento total ideal de 1000 MIPS para una *pipeline* que opera con un ciclo de 1 ns.

Esta sección termina con un análisis idealizado que demuestra algunas de las trampas del *pipelining* profundo. Suponga, por cuestiones de argumento, que la latencia total t ns de una instrucción se puede dividir en cualquier número arbitrario σ de etapas de *pipeline*, con una cabecera de tiempo de τ ns por etapa. Lo anterior conduce a una latencia de etapa *pipeline* de $t/\sigma + \tau$ ns, o una tasa de reloj de $1/(t/\sigma + \tau)$ GHz. Más aún, suponga que una decisión de bifurcación se puede hacer cerca del punto medio de la *pipeline*, de modo que para cada bifurcación que se tome se deben insertar casi $\sigma/2$ burbujas en ella. Finalmente, suponga que tomar bifurcaciones constituye una fracción b de todas las instrucciones que se ejecutan. Con estas suposiciones, el número promedio de ciclos por instrucción es:

$$\text{CPI} = 1 + \frac{b\sigma}{2}$$

Entonces, el rendimiento total promedio es

$$\text{Rendimiento total} = \frac{\text{Tasa de reloj}}{\text{CPI}} = \frac{1}{(t/\sigma + \tau)(1 + b\sigma/2)} \text{ Instrucciones/ns}$$

Al calcular la derivada del rendimiento total con respecto a σ e igualándola a 0, se encuentra el número óptimo de etapas *pipeline*:

$$\sigma^{\text{opt}} = \sqrt{\frac{2t/\tau}{b}}$$

Con base en el análisis precedente, el número óptimo de etapas *pipeline* aumenta conforme la razón t/τ aumenta, y disminuye con un aumento en la fracción b . Diseñar una *pipeline* con más de σ^{opt} etapas sólo dañaría el rendimiento.

PROBLEMAS

15.1 Pipeline de registro de estudiantes

Suponga que en la *pipeline* de registro de estudiantes (figura 15.1), la latencia en cada ventanilla no es constante, sino que varía de uno a dos minutos. Cuando un estudiante llega a una ventanilla específica, permanece ahí hasta que la siguiente ventanilla queda disponible. El movimiento de una ventanilla a la siguiente es instantáneo (es decir, puede ignorar esta cabecera).

- ¿Cómo se debe usar esta *pipeline* para maximizar su rendimiento total?
- ¿El método de la parte a) es aplicable a una ruta de datos de procesador?

- ¿Cuál sería el rendimiento total del esquema de la parte a) si la latencia de cada ventanilla estuviese uniformemente distribuida en $[1, 2]$?
- Discuta un método para mejorar el rendimiento total de esta *pipeline* más allá del obtenido en la parte c) (obviamente, debe modificar una o más de las suposiciones).

15.2 Fotocopiadora encauzada

La fotocopiadora del ejemplo 15.1 tiene dos alimentadores de documento y dos bandejas de salida, de modo que uno de aquéllos puede llenarse y una de éstas puede

vaciarse mientras el otro par está en uso. Esto último elimina la cabecera de carga y descarga de 15 s. Grafique, como funciones de x , la aceleración de esta nueva fotocopiadora y la copiadora original del ejemplo 15.1, en relación con una copiadora no encauzada, y discuta las diferencias observadas. Relacione sus hallazgos con la ley de Amdahl.

15.3 Representación gráfica de *pipelines*

La figura 15.3 muestra dos representaciones diferentes de una *pipeline* de cinco etapas, y grafica instrucciones contra ciclos y etapas contra ciclos. Discuta si la tercera combinación, etapas contra ciclos, conduce a una representación útil.

15.4 Conceptos de ruta de datos encauzada

Considere la ruta de datos encauzada de cinco etapas para MicroMIPS (figura 15.9).

- Explique qué ocurre en cada etapa para las instrucciones *load*, *store*, *add* y *jump*.
- ¿Qué se almacena en cada uno de los registros *pipeline* en la frontera entre etapas consecutivas?
- ¿Los cachés de instrucción y de datos se pueden fusionar en una sola unidad? Si fuera así, ¿cuál sería el efecto de esta acción sobre la unidad de control? Si no lo fuera, ¿por qué?

15.5 Dependencias y burbujas *pipeline*

- Identifique todas las dependencias de datos en la siguiente secuencia de instrucciones, cuando se ejecuta en la implementación MicroMIPS encauzada de la figura 15.9.

```

      addi    $9,$zero,0
      addi    $12,$zero,5000
Loop: addi    $12,$12,4
      lw      $8,40($12)
      add     $9,$9,$8
      addi    $11,$11,-1
      bne     $11,$zero,Loop

```

- Determine el número y lugares de las burbujas que se deben insertar para ejecutar correctamente la secuencia de instrucciones de la parte a). Explique su razonamiento en cada paso.
- ¿Puede sugerir algún reordenamiento de instrucciones que reduciría el número de burbujas?

15.6 Dependencias y burbujas *pipeline*

Repita el problema 15.5 para cada una de las secuencias de instrucciones que aparecen en:

- Solución al ejemplo 5.3
- Solución al ejemplo 5.4
- Solución al ejemplo 5.5
- Problema 5.11
- Solución al ejemplo 6.2
- Figura 7.2

15.7 Rendimiento de ruta de datos encauzada

Considere los siguientes *ciclos* ejecutados en la ruta de datos encauzada de cinco etapas para MicroMIPS (figura 15.9). ¿Cuál de éstos usa la *pipeline* de modo más eficiente y por qué? Establezca claramente todas sus suposiciones.

- Copiar los contenidos del arreglo A en el arreglo B
- Recorrer una lista vinculada linealmente mientras cuenta sus elementos
- Recorrer un árbol de búsqueda binario para hallar un elemento deseado

15.8 Efectos de la mezcla de instrucciones sobre *pipelining*

Considere la mezcla de instrucciones supuesta al final de la sección 15.3 (que se usa en el ejemplo 15.3). Ahora considere una segunda mezcla de instrucciones de “cálculo intensivo” derivada de aquella mediante el aumento de la fracción de tipo R a 56% y disminución de la fracción *branch* a 6%. Calcule el CPI efectivo para la nueva mezcla de instrucciones y compare el rendimiento de la *pipeline* en los dos casos.

15.9 Opciones de selección en ruta de datos encauzada

Considere el par de multiplexores de tres entradas que aparece en la etapa 2 de la *pipeline* que se muestra en la figura 15.9. Para cada una de las nueve posibles combinaciones de establecimientos de estos multiplexores:

- Indique si alguna vez se usa y, si es así, para ejecutar cuál(es) instrucción(es) MicroMIPS.
- Mencione los posible establecimientos de los multiplexores en las etapas 4 y 5 que tendrían sentido.

15.10 Ruta de datos encauzada

Uno de los cuatro cambios en la ruta de datos encauzada de la figura 15.9 en relación con la contraparte no encauzada de la figura 13.3, según se destaca al comienzo de la sección 15.4, es evitable. ¿Cuál es y cuáles son los posibles inconvenientes de no hacer dicho cambio?

15.11 Contenidos de registros *pipeline*

Repita el ejemplo 15.4, pero use las secuencias de instrucciones que aparecen a continuación. Realice suposiciones razonables acerca de los contenidos de registro y memoria donde sea necesario.

- Solución al ejemplo 5.3.
- Solución al ejemplo 5.4.
- Solución al ejemplo 5.5.
- Problema 5.11.
- Solución al ejemplo 6.2.
- Figura 7.2.

15.12 Contenidos de registro *pipeline*

En el ejemplo 15.4 se dice que las entradas en blanco de la tabla 15.1 asociada son desconocidas. En realidad, se pueden especificar algunas de estas entradas sin conocer las instrucciones subsecuentes. ¿Cuáles son y por qué?

15.13 Control encauzado

Repita el ejemplo 15.5, pero use las secuencias de instrucciones que aparecen a continuación. Realice suposiciones razonables acerca de los contenidos de registro y memoria donde sea necesario.

- Solución al ejemplo 5.3.
- Solución al ejemplo 5.4.
- Solución al ejemplo 5.5.
- Problema 5.11.
- Solución al ejemplo 6.2.
- Figura 7.2.

15.14 Alternativas de *pipelining*

- Suponga que, en la pipeline que se muestra en la figura 15.11, la ALU ocupa tres etapas en vez de dos. Discuta los efectos de las dependencias de datos y

control en términos del número mínimo de burbujas o rendijas de retardo necesarios.

- Repita la parte a), esta vez suponiendo una ALU de dos etapas y memorias caché de tres etapas (para ambas cachés).

15.15 Rendimiento de *pipelines* alternas

Con las suposiciones del ejemplo 15.6, compare el rendimiento de las implementaciones encauzadas de MicroMIPS de nueve y diez etapas definidas en el problema 15.14, con las de las anteriores *pipelines* de cinco y ocho etapas.

15.16 *Pipelines* no lineales o bifurcadas

En la analogía de *pipelining* de la figura 15.1, suponga que se encuentra que un estudiante debe pasar el doble de tiempo en la ventanilla que registra, que en cualquiera otra ventanilla. Una solución, sugerida en la sección 15.6, es dividir la función de dicha ventanilla en dos subfunciones, lo que aumenta el número de etapas *pipeline* a seis. Suponga que tal subdivisión no es posible porque la función no se puede subdividir de manera adecuada.

- Muestre cómo proporcionar dos ventanillas de registro puede ayudar a resolver el problema.
- Describa la operación de la nueva *pipeline*, y demuestre que su rendimiento total sería el mismo que el correspondiente a la primera solución sugerida.
- ¿La idea sugerida en la parte a) es aplicable a una ruta de datos de procesador? ¿Cómo o por qué no?

15.17 *Pipelines* no lineales o bifurcadas

En la discusión del *pipelining* óptimo, al final de la sección 15.6:

- ¿Cómo cambiaría el resultado si la decisión *branch* se pudiese hacer cerca del punto de un tercio o de el punto de un cuarto de la *pipeline*?
- Se entiende de manera intuitiva que σ^{opt} aumenta con un incremento en t/τ , o con una disminución en b . Proporcione una explicación intuitiva para la relación no lineal entre σ^{opt} y las dos entidades t/τ y $1/b$.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Crag96] Cragon, H. G., *Memory Systems and Pipelined Processors*, Jones and Bartlett, 1996.
- [Flynn95] Flynn, M. J., *Computer Architecture: Pipelined and Parallel Processor Design*, Jones and Bartlett, 1995.
- [Hart02] Hartstein, A. y T. R. Puzak, “The Optimum Pipeline Depth for a Microprocessor”, *Proceedings of the 29th International Symposium on Computer Architecture*, mayo de 2002, pp. 7-13.
- [Spra02] Sprangle, E. y D. Carmean, “Increasing Processor Performance by Implementing Deeper Pipelines”, *Proceedings of the 29th International Symposium on Computer Architecture*, mayo de 2002, pp. 25-34.

■ CAPÍTULO 16

LÍMITES DEL RENDIMIENTO DE PIPELINE

“Parecería que hemos alcanzado los límites de lo que es posible lograr con la tecnología de computadoras, aunque se debe ser cuidadoso con tales afirmaciones, pues parecerán bastante tontas en cinco años.”

John von Neumann, circa 1949

“Precaución: La capa no permite volar al usuario.”

Etiqueta de advertencia en un disfraz de Batman

TEMAS DEL CAPÍTULO

- 16.1** Dependencias y riesgos de datos
- 16.2** Adelantamiento de datos
- 16.3** Riesgos de la bifurcación *pipeline*
- 16.4** Predicción de bifurcación
- 16.5** *Pipelining* avanzado
- 16.6** Excepciones en una *pipeline*

Las dependencias de datos y control evitan que una *pipeline* logre su pleno rendimiento potencial. En este capítulo se discute cómo una combinación de provisiones de hardware (adelantamiento de datos, predicción de bifurcación) y métodos de software/compilador (bifurcación retardada, reordenamiento de instrucciones) pueden ayudar a recuperar la mayor parte del rendimiento que se podría perder en tales dependencias. También se observarán métodos de encauzamiento (*pipelining*) más avanzados, incluidos las *pipelines* (tuberías) no lineales y las consideraciones de calendarización asociadas con ellos. El capítulo concluye con la discusión de cómo se pueden manipular las excepciones que surgen de una instrucción sin contaminar otras instrucciones parcialmente ejecutadas.

■ 16.1 Dependencias y riesgos de datos

En la sección 15.2 se aprendió que la dependencia de datos es el fenómeno que se da cuando una instrucción requiere datos generados por una instrucción previa. Éstos pueden residir en un registro o localidad de memoria, donde la instrucción subsecuente los encontrará. La figura 16.1 muestra varios ejemplos de dependencias de datos, donde cada una de la segunda a la quinta instrucciones lee un registro en el que escribió la primera instrucción. La quinta de ellas necesita el contenido del registro \$2 después de la conclusión de la escritura del registro por la primera instrucción. Así, esta dependencia

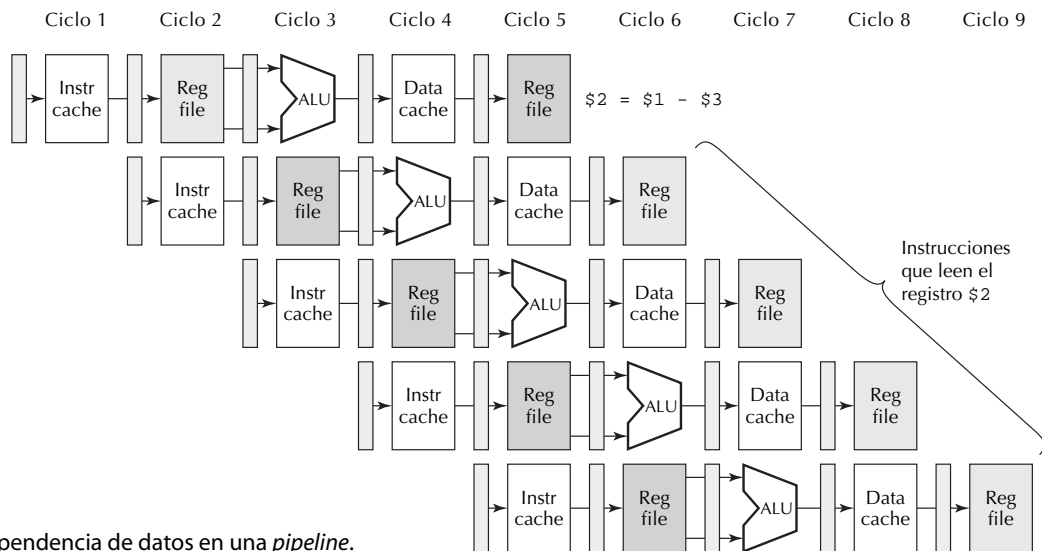


Figura 16.1 Dependencia de datos en una *pipeline*.

de datos no presenta problemas. La cuarta instrucción requiere el nuevo contenido del registro $\$2$ en el mismo ciclo que la primera instrucción lo produce. Ello puede o no presentar un problema, según el diseño del archivo de registro. La segunda y tercera instrucciones necesitan el contenido del registro $\$2$ antes de que esté disponible por la etapa de escritura de la primera instrucción. ¡Ambos casos son problemáticos!

Una forma de lidiar con estos problemas consiste en que el ensamblador o compilador inserte un número adecuado de burbujas en la *pipeline* para garantizar que ningún registro se lea antes de que su valor actualizado se escriba. Esto implica la inserción de 0-3 burbujas, dependiendo de la ubicación de la instrucción que usa el valor de un registro respecto de otro previo que escribe en dicho registro.

En lo que sigue, considere los dos tipos posibles de dependencia de datos en la implementación encauzada de MicroMIPS:

- La dependencia *leer-después de calcular* existe cuando una instrucción actualiza un registro con un valor calculado y una instrucción subsecuente usa el contenido de dicho registro como operando.
- La dependencia *leer-después de cargar* surge cuando una instrucción carga un nuevo valor de la memoria en un registro y una instrucción subsecuente usa el contenido de dicho registro como operando.

En ambas situaciones, el peor caso ocurre cuando al registro lo lee la instrucción enseguida de la que modificó su contenido.

La figura 16.2 muestra las dependencias leer-después de calcular entre una instrucción que actualiza el registro $\$2$ con el resultado de una operación resta y las tres instrucciones siguientes, cada una usa el registro $\$2$ como operando. Al considerar el peor caso de la instrucción que sigue inmediatamente, se nota que es necesario:

- 3 burbujas si un registro se debe leer un ciclo después de que se actualiza
- 2 burbujas si el registro se puede escribir y leer en el mismo ciclo
- 0 burbujas con adelantamiento de datos

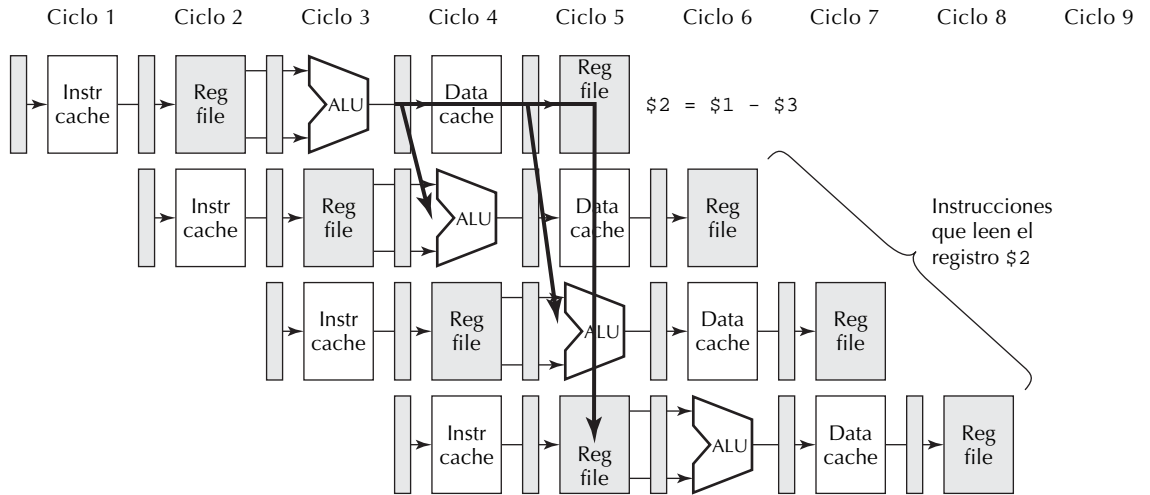


Figura 16.2 Cuando una instrucción previa escribe un valor calculado por la ALU en un registro, la dependencia de datos siempre puede resolverse mediante adelantamiento.

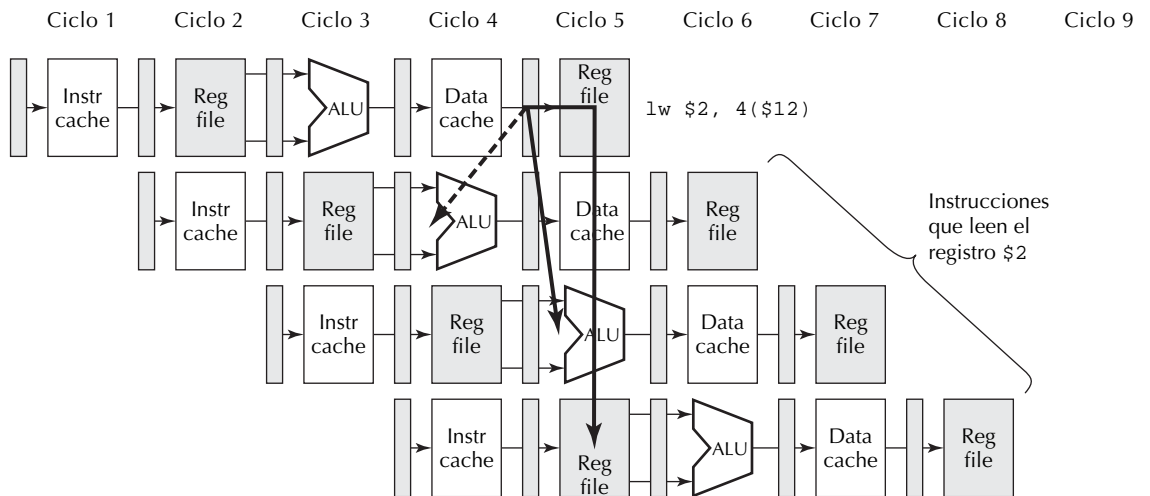


Figura 16.3 Cuando la instrucción inmediatamente anterior escribe en un registro un valor leído de la memoria de datos, la dependencia de datos no se puede resolver mediante adelantamiento (es decir, no se puede retroceder en el tiempo) y en la *pipeline* se debe insertar una burbuja.

En la sección 16.2 se verá que un mecanismo simple permite adelantar el resultado calculado de una instrucción a las instrucciones subsecuentes que necesitan el valor actualizado antes de que se escriba en el registro de destino. Por tanto, las dependencias de datos leer-después-de-calcular no son problemáticas en MicroMIPS encauzado porque no conducen a la degradación del rendimiento.

La figura 16.3 bosqueja las dependencias leer-después-de-cargar entre la carga del registro \$2 y las tres instrucciones siguientes, cada una de ellas usa el registro \$2 como operando. Al considerar el peor de los casos de la instrucción inmediata, se observa que es necesario:

- 3 burbujas si el registro se debe leer un ciclo después de que se escribe.
- 2 burbujas si el registro se puede escribir y leer en el mismo ciclo.
- 1 burbuja con adelantamiento de datos.

En consecuencia, con el mecanismo de adelantamiento a discutir en la sección 16.2, el compilador debe insertar una sola burbuja en la *pipeline* sólo cuando una instrucción *load* es seguida por una instrucción que usa el registro cargado como operando. Con frecuencia, tal situación se evita al mover otra instrucción entre la *load* y su instrucción dependiente de datos siguiente. Por tanto, con el diseño adecuado del compilador, las dependencias de datos leer después de cargar no presenta serios problemas en MicroMIPS encauzado, porque tienen un efecto mínimo en el rendimiento.

Para resumir, las dependencias de datos en procesadores encauzados se pueden manipular mediante software, hardware o métodos híbridos. Una solución pura de software permite que el compilador asegure que las instrucciones dependientes de datos están bastante espaciadas para garantizar la disponibilidad de cada elemento de datos antes de su uso real. El compilador manipula esto último mediante el reordenamiento de instrucciones, cuando es posible, o a través de la inserción de burbujas. Una solución pura de hardware requiere la incorporación de un mecanismo de adelantamiento de datos que intercepte valores de datos ya disponibles en la ruta de datos, pero todavía no almacenadas adecuadamente en un registro, y las envía a las unidades que las requieren. Cuando el adelantamiento no es posible, el hardware bloquea la instrucción dependiente hasta que sus datos requeridos se suministren. La última opción se conoce como *interbloqueo de tubería (pipeline interlock)* y equivale a la inserción de burbujas en tiempo de ejecución cuando es necesario. Una solución híbrida usa una mezcla de métodos hardware y software que se estiman más efectivos en costo; por ejemplo, el adelantamiento de datos es útil para las dependencias más comunes, que además no complican el hardware (más bien lo frenan), y el manejo de las instancias raras o complejas se relega al software.

■ 16.2 Adelantamiento de datos

El adelantamiento de datos obvia la necesidad de insertar burbujas en la *pipeline*; por tanto, reduce o elimina las pérdidas asociadas en el rendimiento, a través de enrutar (*routing*) los resultados de instrucciones parcialmente completadas hacia donde necesiten instrucciones subsecuentes. En la ruta de datos MicroMIPS encauzada de la figura 15.9, considere una instrucción I2 que acabe de completar su fase de lectura de registro en la etapa (*stage*) 2 y esté por moverse hacia la etapa 3. En este instante, las instrucciones I3 e I4 están por delante de dicha instrucción, y sus resultados parciales y señales de control se almacenan al final de las etapas 3 y 4, respectivamente. La tarea de la unidad de adelanto (figura 16.4) consiste en determinar si alguna de estas dos instrucciones escribirá un valor en los registros *rs* y *rt* cuyos contenidos apenas se leyeron en los registros *x2* y *y2* y, si es así, para adelantar el valor más reciente a la ALU en lugar de *x2* o *y2*.

Por ejemplo, considere *x2*, que se cargó de *rs*, con el índice *rs* salvado como *s2* en el registro *pipeline*. A partir de las señales *RegWrite3* y *RegWrite4*, se sabe si alguna de las instrucciones I3 o I4 escribirá algo en un registro. Si ninguna instrucción escribirá en un registro, entonces no hay razón para preocuparse y *x2* se puede dirigir a la ALU. Si alguna instrucción tiene postulada su señal *RegWrite*, entonces una comparación de sus índice registro (*d3* o *d4*) con *s2* indicará si el registro *rs* que se acaba de leer será sobrescrito por una instrucción parcialmente calculada. Si fuera así, uno de los valores *x3*, *y3*, *x4* o *y4* se debe dirigir a la ALU en lugar de *x2*. Es fácil señalar cuál, al tomar en cuenta los resultados de las dos comparaciones *s2matchesd3* (postulada si $s2 = d3$) y *s2matchesd4* (postulada si $s2 = d4$), así como las cinco señales de control almacenadas al final de las etapas 3 y 4. En la tabla 16.1 se proporciona una tabla de verdad parcial para la unidad de adelantamiento superior

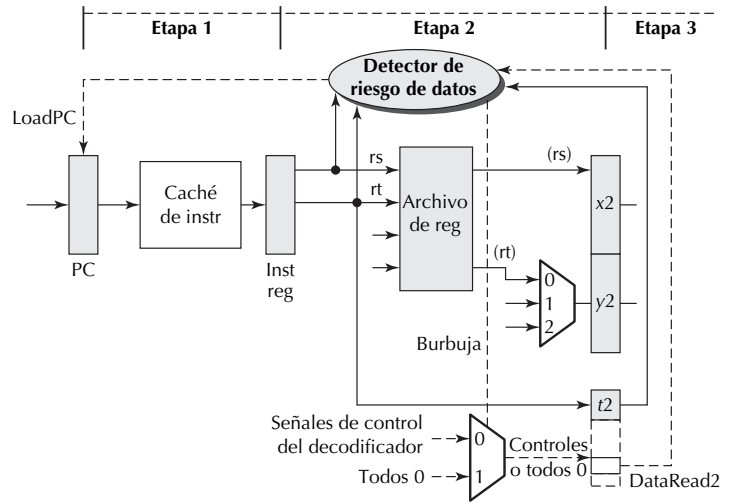


Figura 16.5 Detector de riesgo de datos para la trayectoria de datos MicroMIPS encauzada.

El detector de riesgo de datos tiene una tarea simple. Debe examinar la instrucción al final de la etapa 2 de la *pipeline* para determinar si se trata de una instrucción *load* (se satisface la señal *DataRead2*). Si se trata de una instrucción *load*, y el registro t_2 , en el que escribirá, iguala a cualquier registro rs o rt de la instrucción al final de la primera etapa, la última entrada de instrucción se nulifica (regresa a una *no-op*) al establecer todas las señales de control a 0 conforme la instrucción avanza a la siguiente etapa (el mux en el fondo de la figura 16.5) y la señal *LoadPC* se despostula de modo que el PC no se actualiza. Esto conduce a que la misma instrucción se lea (*fetch*) una segunda vez y se ejecute con el requisito de un ciclo de retardo. Por tanto, la instrucción *no-op* insertada es la burbuja de *pipeline* que resuelve el problema de dependencia de datos. En todos los otros casos, la señal *Bubble* se despostula y la señal *LoadPC* se postula, lo que provoca que la siguiente instrucción se lea (*fetch*) como siempre.

Note que, en la ruta de datos encauzada original de la figura 15.9, no hay señal *LoadPC* porque el PC se carga en cada ciclo de reloj. Por ende, la ruta de datos encauzada original no funcionaría correctamente a menos que el compilador o ensamblador inserte automáticamente una burbuja (*no-op*) después de cada instrucción *load* que está seguida inmediatamente por una instrucción que use el valor apenas cargado. Cualquier instrucción que no use el resultado de la instrucción *load* puede seguirla en la *pipeline* sin poner en peligro la operación correcta. En consecuencia, el compilador tiene la opción de mover una instrucción que sea independiente tanto de *load* como de su instrucción dependiente entre las dos, lo que por tanto evita efectivamente la pérdida de una rendija de instrucción a la *no-op*.

Note que el esquema de inserción de burbuja de la figura 16.5 es un poco pesimista: aun cuando toda instrucción cause que los contenidos de rs y rt se lea, no todas las instrucciones realmente usan los valores así obtenidos.

■ 16.3 Riesgos de la bifurcación *pipeline*

Recuerde que toda instrucción *branch* debe estar seguida inmediatamente por una burbuja en la *pipeline* MicroMIPS. Esto es así porque las decisiones de bifurcación y la formación de dirección ocurren en la etapa 2 de la *pipeline* (vea figura 15.9). La instrucción a leer (*fetch*) en la etapa 1 mientras que se realiza una decisión de bifurcación en la etapa 2 puede, por tanto, no ser la instrucción justa. Sin embargo, insertar una burbuja después de cada *branch* es innecesariamente pesimista porque, con frecuencia, la condición de bifurcación no se señala (*DataRead2*). Por ejemplo, una bifurcación que

se usa dentro de un *ciclo* para salir de éste cuando se encuentra una condición de terminación se toma sólo una vez y no se toma para las iteraciones restantes (que pueden ser miles). Insertar *no-op* después de tal instrucción *branch* conduce a una pérdida innecesaria de rendimiento.

Las soluciones para este caso, como el problema de riesgo de datos, caen en las categorías basadas en software y hardware. Como ya se anotó, una solución popular basada en software consiste en cambiar la definición de la bifurcación para incluir la ejecución de la instrucción inmediata antes de que el control se transfiera realmente a la dirección blanco de bifurcación. Es decir, bajo tal esquema de *bifurcación retardada*, si la ruta de dato es tal que la decisión de bifurcación se hace en la etapa β , entonces las $\beta - 1$ instrucciones después de *branch* siempre se ejecutan antes de que ocurra transferencia de control. Para el caso de que existan instrucciones antes de la *branch*, que sean independientes de ella y no alteren esa condición, se pueden mover a las *rendijas de retardo de bifurcación* que siguen a la instrucción *branch* para evitar desperdicio de estas rendijas. En el peor de los casos, si el compilador o programador no pueden identificar algún trabajo útil que se realice en algunas rendijas de retardo de bifurcación, los llenará con *no-ops* como último recurso. En la práctica, esta opción rara vez es ejercida.

En MicroMIPS se toma este enfoque basado en software, y se redefine la bifurcación como retardada con una rendija de retardo de bifurcación. Puede tardar en acostumbrarse a esto si siempre pensó que una instrucción *branch* se efectuaba de inmediato. Sin embargo, en virtud de que los programas en lenguaje de máquina son generados predominantemente por compiladores, ésta no resulta una carga irracional. En raros casos, cuando es necesario codificar a mano un programa o procedimiento en lenguaje ensamblador, puede comenzar por insertar automáticamente una *no-op* después de cada bifurcación y, cerca del final del proceso de diseño, buscar instrucciones que se puedan mover hacia dichas rendijas de retardo. Desde luego, si existiera más de una rendija de retardo por llenar, la tarea puede resultar más difícil para el compilador o programador, y quizá aumente la probabilidad de que ciertas rendijas de retardo queden sin uso. Pero esto no debe preocupar en relación con MicroMIPS.

La misma consideración se aplica a las tres instrucciones *jump*, aunque para *j* y *jal*, puede ser factible modificar la ruta de datos encauzada de modo que el PC se modifique en la primera etapa justo después de que la instrucción se lee y se determina si es *j* o *jal*. En otras palabras, para estas dos instrucciones *jump*, el PC y el registro *pipeline* entre la primera y segunda etapas se escribirán al mismo tiempo, ello permite que la instrucción en la dirección destino del salto siga la instrucción *jump* sin retardo alguno.

Las soluciones basadas en hardware extienden el detector de riesgo de datos de la figura 16.5 para detectar y lidiar con riesgos de *branch* (y posiblemente *jump*). Observe que si la instrucción en la etapa 2 no es una *branch*, o sí lo es pero la condición *branch* no se satisface, entonces todo está bien y el detector de riesgo extendido de la figura 16.5 no necesita hacer nada. Por otra parte, si la instrucción es una *branch* y la condición *branch* se satisface, entonces la instrucción apenas leída (*fetch*) en la etapa 1 se debe nulificar. Esto lo puede lograr el detector de riesgo (que ahora trata con riesgos tanto de datos como de *branch*) al postular una señal ClearIR para restablecer el registro de instrucción a todos 0, que representa una instrucción *no-op*. De nuevo, si más de una instrucción se afectase debido a las decisiones *branch* tomadas en la etapa β , donde $\beta > 2$, entonces $\beta - 1$ de los registros *pipeline* interetapas tendrían que restablecerse a todos 0, ello corresponde a la inserción de $\beta - 1$ burbujas.

La estrategia discutida en el párrafo anterior se puede ver como una forma rudimentaria de predicción de bifurcación (sección 16.4). En esencia, la ejecución de instrucción continúa mediante la predicción o suposición de que la bifurcación no se tomará. Si esta predicción resulta correcta, entonces no se paga penalización por rendimiento; de otro modo, la siguiente instrucción, que de cualquier forma no es la que debería ejecutarse, se nulifica y su espacio en la *pipeline* se convierte en una burbuja. En este sentido, se puede ver este enfoque como una estrategia de inserción de burbuja dinámica o de tiempo de ejecución, en oposición a estático o de tiempo de compilación.

■ 16.4 Predicción de bifurcación

La mayoría de los procesadores modernos tienen alguna forma de predicción de bifurcación; esto es, un método para predecir si una bifurcación ocurrirá o no y para leer (*fetch*) instrucciones de la ruta más probable mientras la condición *branch* se evalúa. Este enfoque reduce la probabilidad de que se necesite limpiar la *pipeline*, lo que aumenta el rendimiento. Las estrategias de predicción de bifurcación varían de muy simples a muy complejas (y más precisas). Algunos ejemplos de estrategia son:

1. Siempre predecir que la bifurcación no se tomará, limpiar las instrucciones leídas (*fetch*) si se establece de otro modo. Esto es lo que se ha hecho hasta el momento.
2. Usar el contexto del programa para decidir cuál ruta es más probable, quizá se tome una bifurcación al final de un *ciclo* (*bifurcación hacia atrás*), mientras que una al principio (*bifurcación hacia adelante*) usualmente no se toma. Las bifurcaciones que se usan en los constructos si-entonces-sino son más difíciles de predecir.
3. Permitir al programador o compilador proporcionar pistas de la ruta más probable y codificar éstas en la instrucción.
4. Decidir con base en la historia pasada, mantener una pequeña tabla histórica del comportamiento de cada instrucción *branch* en el pasado y usar esta información para decidir cuál alternativa es más probable. Algunas implementaciones de este método se detallarán más adelante en esta sección.
5. Aplicar una combinación de factores: los modernos procesadores de alto rendimiento usan esquemas muy elaborados para predicción de bifurcación porque con *pipelines* profundas (*super-pipelining*) la penalización por una predicción equivocada es muy elevada.

Enfóquese en el cuarto método, o basado en la historia. Se trata de un ejemplo de una estrategia de *predicción dinámica de bifurcación*, en contraste con los métodos de *predicción estática de bifurcación* que se apoyan en la instrucción *branch per se* o en su contexto de programa. La mínima cantidad de historia es, desde luego, un solo bit que indica si se tomó o no una instrucción *branch* particular la última vez que se ejecutó. Esta estrategia muy simple es precisa para bifurcaciones que controlan repeticiones de *ciclo* o salidas; el primer tipo casi siempre se toma, y la única excepción se encuentra en la última iteración del *ciclo*, mientras que la última sólo se toma una vez. En consecuencia, en un *ciclo* que se ejecuta 100 veces, esta estrategia de predicción sería 98% a 99% precisa. Las bifurcaciones asociadas con los constructos si-entonces-sino no son tan uniformes; por ende, no se pueden predecir con tanta precisión.

Observe que el uso de un solo bit de historia provoca dos malas predicciones para cada instancia de un *ciclo*. Por ejemplo, considere una bifurcación hacia atrás al final de un *ciclo*. Esta bifurcación se predecirá mal una vez al final de la primera iteración (porque la última vez no se tomó) y de nuevo al final de la última iteración. Esta penalización se puede cortar a la mitad (es decir, reducirla a sólo una mala predicción por ejecución de ciclo) al no permitir un solo cambio de comportamiento para alterar la predicción. La figura 16.6 muestra un esquema de predicción de bifurcación de cuatro estados que corres-

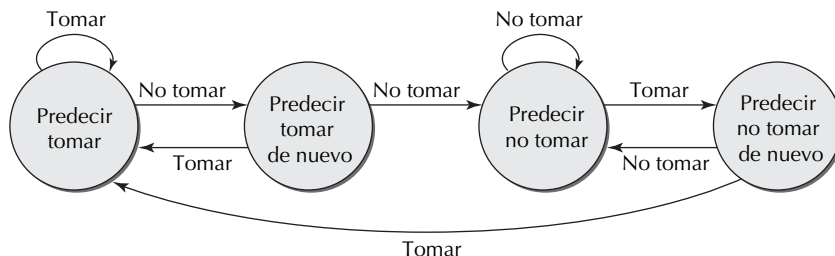


Figura 16.6 Esquema de predicción de bifurcación de cuatro estados.

ponde a mantener dos bits de historia. En tanto continúa tomándose una bifurcación, se predice que se tomará la siguiente vez (el estado de la extrema izquierda en la figura 16.6). Después de la primera mala predicción el estado cambia, pero continúa la predicción de que la bifurcación se tomará. Una segunda mala predicción causa otro cambio de estado, esta vez a un estado que causa la predicción opuesta.

Ejemplo 16.1: Comparación de estrategias de predicción de bifurcación Un programa consta de dos ciclos anidados, con una sola instrucción *branch* al final de cada *ciclo* y ninguna otra instrucción *branch* en otra parte. El *ciclo* exterior se ejecuta 10 veces y el interior 20. Determine la precisión de las siguientes tres estrategias de predicción de bifurcación: *a)* siempre predecir tomar, *b)* usar un bit de historia, *c)* usar dos bits de historia según la figura 16.6.

Solución: Desde el punto de vista de la predicción de bifurcación, se tienen 210 iteraciones de *ciclo* o ejecuciones de instrucción *branch*: $10 \times 20 = 200$ para el *ciclo* interior (190 tomar, 10 no tomar) y 10 para el *ciclo* exterior (nueve tomar, una no tomar). La opción *a)* produce 11 malas predicciones y tiene una precisión de $199/210 = 94.8\%$. La opción *b)* tiene una mala predicción para el *ciclo* exterior y 19 para el ciclo interior (una vez en su primera ejecución y dos veces por cada ejecución subsecuente), si se supone que la historia de un bit para cada ciclo se inicializa como “tomar”. La precisión en este caso es $190/210 = 90.5\%$. La opción *c)* conduce a la misma respuesta que la opción *a)*. Se ve que, en este ejemplo, la estrategia más simple también es la mejor; sin embargo, esto sucede porque todas las bifurcaciones son del mismo tipo hacia atrás.

Tales estrategias dinámicas de predicción de bifurcación se pueden hacer muy elaboradas. El esquema más elaborado es más preciso, pero también involucra más cabeceras en términos de complejidad de hardware. La cabecera de tiempo de la predicción de bifurcación por lo general no es un conflicto, porque se puede remover de la ruta crítica de ejecución de instrucción. Conforme las *pipelines* de procesador se hacen más profundas, el uso de rendijas de retardo de bifurcación se favorece menos y la confiabilidad se coloca en estrategias de predicción de bifurcación más precisas.

Considere el siguiente ejemplo, que usa varias ideas para predicción de bifurcación más precisa y *prefetching* más rápida de la ruta predicha (figura 16.7). Unos cuantos bits de orden inferior de la dirección en PC se usan para indexar una tabla que almacena información acerca de la última instrucción *branch* encontrada que tenga los mismos bits de dirección de orden inferior. Esta información incluye la dirección completa de la instrucción *branch*, su dirección destino y bits de historia, como se discutió antes. Si los bits de historia indican que quizá se tomará la bifurcación, la dirección destino de bifurcación almacenada inmediatamente se carga en el PC sin esperar el cálculo de dirección ni la decodificación de la instrucción.

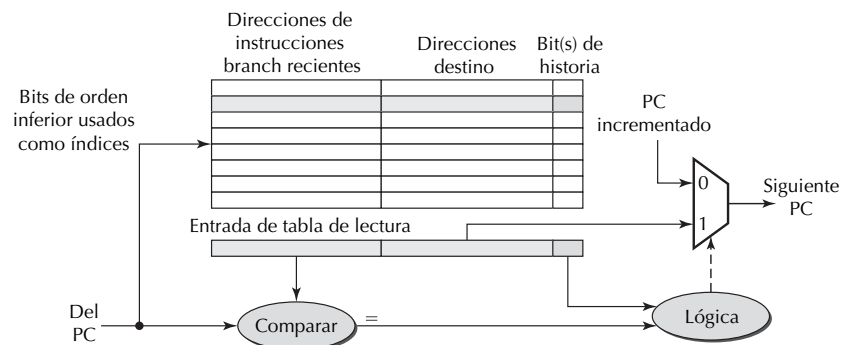


Figura 16.7 Elementos de hardware para un esquema de predicción de bifurcación.

■ 16.5 *Pipelining* avanzado

La mayoría de los procesadores actuales de alto rendimiento son superencauzados (*superpipelined*). Sin embargo, existe un límite a la mejora de rendimiento a través de *pipelines* más profundas. Las *pipelines* más profundas implican mayor cabecera. Conforme las etapas se hacen más rápidas, la cabecera de *latching* y el margen de seguridad permitido para sesgo de reloj y otras incertidumbres comprometen una fracción significativa del tiempo empleado en la realización de cálculos. Esta cabecera, combinada con la penalización más severa de limpieza de *pipeline* y el requisito de un mayor número de burbujas debido a dependencias de datos y control, coloca un límite superior sobre la profundidad de la *pipeline*. Actualmente las *pipelines* de 20-30 etapas están en la frontera de lo practicable. Además de las dependencias de los tipos ya discutidos, los fallos de caché (capítulo 18) infligen incluso una penalización de rendimiento mayor. Una instrucción que encuentra un fallo en caché de datos no avanza hasta que los datos requeridos se llevan al caché. Esto último “atasca” la *pipeline* durante decenas de ciclos, y en ocasiones empequeñece la penalización por mala predicción de bifurcación.

Ejemplo 16.2: CPI efectivo con fallos de bifurcación y caché En una *superpipeline* de 20 etapas se deben insertar cuatro burbujas para instrucciones *branch* condicionales, ello constituye 15% de todas las instrucciones ejecutadas. Casi 2% de las instrucciones encuentran un fallo de caché cuando acceden a la memoria de datos, lo que causa que la *pipeline* se “atasque” durante 25 ciclos. ¿Cuál es el CPI efectivo para esta *pipeline*?

Solución: Una *pipeline* que siempre está llena (sin burbujas o atascos), conduce a un CPI de 1 a largo plazo. A este CPI de 1 se le debe sumar el número promedio (esperado) de ciclos por instrucción perdidos por burbujas o atascos. Por ende, $\text{CPI promedio} = 1 + 0.15 \times 4 + 0.02 \times 25 = 2.1$. Se ve que más de la mitad del rendimiento teórico máximo de la *pipeline* se pierde en burbujas y atascos.

Dadas las limitaciones en el mejoramiento del rendimiento a través de *pipelining* más profundo mientras se mantiene la estructura lineal y estática de la *pipeline* de procesador, se han diseñado otros métodos arquitectónicos.

El primer método es el uso de bifurcación de *pipelines* en oposición a las estrictamente lineales. La figura 16.8 muestra tal *pipeline* con unidades de función múltiples que pueden ser del mismo tipo o

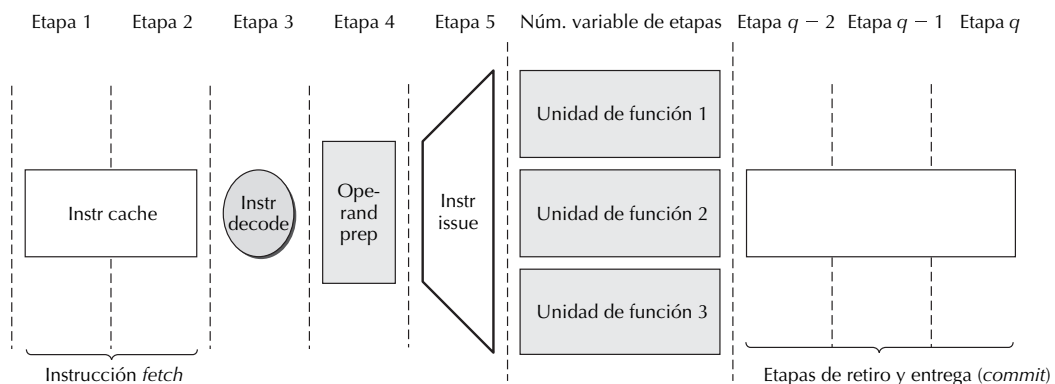


Figura 16.8 Pipeline dinámica de instrucción con emisión en orden, posible conclusión fuera de orden y retiro en orden.

especializadas (por ejemplo, ALU entero y de punto flotante separadas). Al igual que en una *pipeline* lineal, en ausencia de dependencias de datos y control, una instrucción se inicia en cada ciclo de reloj. Diferentes instrucciones pueden tomar distintas rutas en las porciones de bifurcación de la *pipeline* y emerger en el otro extremo del orden. La conclusión fuera de orden quizá se deba a diferente número de etapas en las unidades de función o a distintas latencias dependientes de datos en la misma unidad de función. En consecuencia, por ejemplo, las instrucciones I1, I2, I3, I4 que entran a la *pipeline* se pueden completar en el orden I1, I3, I4, I2. Las etapas de retiro y entrega al final de la *pipeline* se cargan con la tarea de juzgar cuando todas las instrucciones que preceden a una instrucción específica se completaron y sólo entonces entrega los resultados temporales de tal instrucción a almacenamiento permanente (registro o caché). De esta forma, si ocurre una interrupción, las instrucciones parcialmente completadas y las completadas por adelantado de otras no conducirán a inconsistencias en los contenidos de registro o memoria.

Una versión más avanzada del esquema que se muestra en la figura 16.8 permite que la lógica de emisión de instrucción mire por adelantado y las instrucciones emitidas más allá de las conocidas, o esperadas, se “atasquen” en la *pipeline*.

El segundo método prescribe el uso de múltiples *pipelines* independientes que permiten muchas instrucciones a iniciar a la vez (2 a 4 es típico). En tales *procesadores superescalares*, CPI se puede volver menor que 1. Observe un diseño superescalar de dos vías. En tal procesador, el archivo de registro se debe modificar para permitir hasta cuatro lecturas de registro y dos operaciones de escritura de registro a realizar en cada ciclo. De manera alterna, se puede permitir sólo una operación de escritura y restringir las instrucciones emitidas concurrentemente a pares de instrucciones que no escriban ambas en un registro.

Desde luego, la secuencia de instrucciones puede ser tal que la ejecución superescalar no ofrezca beneficio, pero esta situación es rara. Considere el siguiente *ciclo* como ejemplo:

```

Loop: lw    $t0,0($s1)      # carga $t0 con elemento de arreglo
      add   $t0,$t0,$s2     # suma escalar en $s2
      sw    $t0,0($s3)     # salva el resultado en memoria
      addi  $s1,$s1,-4      # disminuye apuntador
      bne   $s1,$zero,Loop  # bifurcación si $s1 no es cero

```

La única concurrencia posible es emitir las instrucciones *sw* y *s2* al mismo tiempo.

```

Ciclo 1: lw
Ciclo 2: addi
Ciclo 3: add
Ciclo 4: sw, bn

```

Si esta situación (cinco instrucciones emitidas en cuatro ciclos de reloj) continuara, resultaría un CPI de 0.8. Más usualmente, ambas rendijas de emisión de instrucción se usan en la mayoría de los ciclos y se obtiene un CPI mucho más cercano a la cota inferior de 0.5.

Con base en lo aprendido hasta el momento, la tabla 16.2 resume el efecto de varias arquitecturas de procesador y estrategias de implementación asociadas en CPI y, por tanto, en rendimiento. La discusión acerca de la arquitectura de procesadores comenzó con la ruta de datos de ciclo sencillo en el capítulo 13 y se procedió a una implementación multiciclo más práctica en el capítulo 14. En el capítulo 15 se introdujo una simple *pipeline* estática. Aun cuando una *pipeline* estática teóricamente puede lograr un CPI de 1, el CPI real con frecuencia está en el rango de 2 a 3 cuando se consideran las penalizaciones debidas a dependencias de datos, malas predicciones de bifurcación y fallos de caché (capítulo 18). El *pipelining* dinámico con ejecución fuera de orden se puede llevar bastante cerca de

■ **TABLA 16.2** Efecto de arquitectura de procesador, métodos de predicción de bifurcación y ejecución especulativa en CPI.

Arquitectura	Métodos usados en la práctica	CPI
No encauzada, multiciclo	Estricta emisión y ejecución de instrucciones en orden	5-10
No encauzada, traslapada	Emisión en orden, con múltiples unidades de función	3-5
Encauzada, estática	Ejecución en orden, predicción simple de bifurcación	2-3
Superencauzada, dinámica	Ejecución fuera de orden, predicción avanzada de bifurcación	1-2
Superescalar	Emisión de dos a cuatro vías, interbloqueo y especulación	0.5-1
Superescalar avanzada	Emisión de cuatro a ocho vías, especulación agresiva	0.2-0.5

un CPI de 1. Para ir más allá de esto, es necesario apoyarse en la emisión múltiple de instrucciones (diseño superescalar) y quizá en la ejecución especulativa más avanzada.

Los métodos basados en software están entrelazados con las técnicas de hardware ya mencionadas para garantizar máximo rendimiento con una arquitectura específica de procesador. En este sentido, conforme evolucionan la profundidad de *pipeline* y otras características de implementación de procesador, se deben introducir nuevas versiones de software para obtener ventaja completa de las nuevas capacidades de hardware. Versiones antiguas pueden “correr” correctamente, pero es improbable que lleven a un rendimiento óptimo.

Una técnica de software que se ha encontrado útil para aumentar la probabilidad de que múltiples instrucciones independientes estén disponibles para ejecución superescalar es el *desenrollado de ciclo*. Considere el siguiente *ciclo* mostrado en forma abstracta:

Ciclo: Hacer ciertos cálculos que involucren el valor índice *i*
Incrementar el índice *i* del *ciclo* por 1
Verificar condiciones de terminación y repetir si se requiere

Desenrollar el *ciclo* una vez resulta en lo siguiente:

Ciclo: Hacer algunos cálculos que involucren el valor índice *i*
Hacer algunos cálculos que involucren el valor índice *i* + 1
Incrementar el índice *i* del *ciclo* por 2
Verificar condiciones de terminación y repetir si se requiere

El *ciclo* desenrollado se ejecutará la mitad de veces, y hará el doble de cálculos en cada iteración. Desde luego, se podría desenrollar el *ciclo* tres veces, cuatro o más. Desenrollar aumenta el número de instrucciones en el texto del programa, pero conduce a una reducción en el número de instrucciones ejecutadas. Lo anterior se debe a la cabecera inferior de verificación de índice y bifurcación en la versión desenrollada. Desenrollar reduce el número de instrucciones *branch* ejecutadas y crea más instrucciones dentro del cuerpo del *ciclo* que pueden ser independientes una de otra y, por tanto, capaces de emitirse juntas en una arquitectura superescalar.

Esta sección termina con un breve panorama del desarrollo arquitectónico de los microprocesadores Intel [Hint01], el más reciente de ellos está en uso hoy día en las computadoras de escritorio y laptops (tabla 8.2). El procesador Intel Pentium, y sus predecesores inmediatos (286, 386, 486), tienen similares profundidades de *pipeline*, y la mejora en el rendimiento se logra mediante avances en la tecnología. El Pentium Pro comenzó una serie de productos que casi duplicó la profundidad de *pipeline* y condujo a los procesadores Pentium II, III y Celeron. Este aumento en profundidad de *pipeline*, y la mejora asociada en frecuencia de reloj (aunque no mucho por un factor de 2) se logró al descomponer las instrucciones IA32 en micro-ops y usar ejecución fuera de orden. Pentium 4 (por alguna razón, Intel abandonó el uso de números romanos para numerar su familia de procesadores Pentium) duplicó

después la profundidad de *pipeline*, ello condujo a la necesidad de especulación más agresiva y muchas mejoras arquitectónicas asociadas.

■ 16.6 Excepciones en una *pipeline*

En un procesador no encauzado, la ocurrencia de una excepción, como un desbordamiento aritmético, causa una transferencia de control a una rutina de sistema operativo (figura 14.10). Las interrupciones se tratan de manera similar en cuanto a que, después de la conclusión de la instrucción actual, el control se transfiere a una rutina de manejo de interrupción. En un procesador encauzado, las excepciones y bifurcaciones condicionales presentan problemas del mismo tipo. Una excepción de desbordamiento equivale a “bifurcación a una dirección especial si ocurre desbordamiento”. Sin embargo, en virtud de que las excepciones son raras, es suficiente asegurar el manejo correcto de la excepción; el rendimiento y otros conflictos relacionados con la predicción de bifurcación no están involucrados.

Cuando los efectos de las excepciones en un procesador encauzado son idénticos a los de un procesador no encauzado con ejecución meramente secuencial, se dice que la implementación encauzada soporta *excepciones precisas*. Una de ellas es deseable porque permite que el programa se reanude más tarde desde el punto de interrupción sin efectos dañinos. Algunas de las primeras máquinas de alto rendimiento tienen *excepciones imprecisas*, ello significa que, en algunos casos, los resultados observados fueron inconsistentes con la ejecución secuencial simple. Eso se hizo por razones de costo y rendimiento, pues el costo de garantizar excepciones precisas, y sus penalizaciones de rendimiento asociadas, no se consideraban justificadas. Las *interrupciones precisas e imprecisas* tienen connotaciones similares. El resto de esta sección se dedica a estudiar las opciones de implementación para excepciones precisas en procesadores encauzados, la discusión de las interrupciones se verá en el capítulo 24.

Cuando ocurre una excepción durante la ejecución de una instrucción en un procesador encauzado, ésta y las subsecuentes en el orden del programa cuyo procesamiento ya inició, se deben limpiar de la *pipeline*, mientras que las instrucciones por adelante de la que conduce a una excepción se deben dejar “correr” para concluir antes de que el control se transfiera a una rutina de manejo de excepción. En una *pipeline* lineal simple, como la que se usa en MicroMIPS, eso se hace mediante la postulación de la señal de control *clear* para cada conjunto de registros *pipeline* interetapas que preceden la etapa en la que ocurrió la excepción. Esta estrategia funciona en tanto ninguna de las instrucciones limpiadas actualice el valor en un registro o localidad de memoria. También es necesario que la rutina de manejo de excepción sepa cuál instrucción condujo a la excepción. Por tanto, el contenido del PC asociado con la instrucción que causó la excepción se pondrá a disposición mediante su escritura en un registro especial o localidad de memoria. Micromips tiene un registro EPC (PC de excepción) especial en el que se escribe el valor PC de la instrucción que produjo la excepción (figura 5.1).

En un procesador con *pipelining* dinámico o emisión múltiple de instrucción por ciclo, el ofrecimiento de excepciones precisas es más complicado. Con referencia a la figura 16.8, se nota que cuando la operación en una de las unidades de función conduce a una excepción, las otras unidades de función pueden estar ejecutando instrucciones que estén por delante, o detrás, de dicha instrucción en la secuencia normal del programa. Las que están por delante se deben dejar “correr” para su conclusión, mientras que las que están por detrás se deben limpiar. Las etapas de retiro y entrega al final de la *pipeline* logran esto en forma natural. Estas etapas retiran instrucciones en estricto orden secuencial de modo que una instrucción no se retira; por tanto, sus resultados no se entregan permanentemente, a menos que las instrucciones previas ya se hayan retirado. En consecuencia, en el momento de retirar la instrucción causante de la excepción, la unidad toma nota de la excepción y no retira instrucciones posteriores; por tanto, los resultados parciales no entregados de tales instrucciones no tendrán efecto sobre el estado permanente del procesador.

La implementación de hardware del mecanismo de retiro y entrega de instrucción puede tomar dos formas:

1. *Archivos de historia*. En el curso de la ejecución de instrucciones actualiza registros con nuevos valores, pero mantiene los valores previos, de modo que cuando debido a una excepción se necesita una regresión, se pueden restaurar los valores antiguos.
2. *Archivos futuros*. No escribe nuevos contenidos de registro en el archivo de registro, los conserva en un archivo futuro y copia estos valores actualizados en el archivo de registro principal en el momento en que todas las instrucciones previas se retiran.

Cuando las latencias de ejecución de varias instrucciones son diferentes, un gran número de instrucciones no retiradas y sus resultados se almacenarán en *buffer* en la unidad de retiro y entrega mientras esperan la conclusión de otras instrucciones. Esto último conduce a un alto costo de implementación y quizás a penalización de rapidez no despreciable. Por esta razón, algunos procesadores ofrecen excepciones precisas como una opción elegible para evitar la penalización de rendimiento asociada cuando pudieran hacerse excepciones imprecisas.

Una alternativa viable a las excepciones precisas basadas en hardware consiste en permitir la rutina de manejo de excepción para finalizar las instrucciones en progreso con el uso de ejecución de software basada en información que se haya salvado o se obtenga de varias partes de la *pipeline*. De esta forma se paga una significativa penalización de rapidez cuando ocurre una excepción, pero ello puede ser aceptable en virtud de que las excepciones son raras.

PROBLEMAS

16.1 Dependencia de datos cargar-después de-almacenar

En la sección 16.1 se discutieron las dependencias de datos leer-después de-calcular y leer-después de-cargar que se relacionan con la garantía de que el operando de un registro proporcionado a cada unidad retiene el último valor escrito en el registro correspondiente. Discuta si la misma preocupación surge para contenidos de memoria. En otras palabras, ¿son necesarios mecanismos para lidiar adecuadamente con dependencias de datos *cargar-después de-almacenar*?

16.2 Dependencia de datos almacenar-después de-cargar

Considere copiar una lista de n palabras de un área en memoria a otra área. Esto se puede lograr mediante la colocación de un par de instrucciones lw y sw en un *ciclo*, y que cada iteración del *ciclo* que copie una palabra. En la implementación actual de MicroMIPS encauzado con adelantamiento, que se bosqueja en las figuras 16.4 y 16.5, esto conduce a una burbuja entre lw y sw . ¿Es posible evitar este atascamiento mediante hardware de adelantamiento de datos adicional? Discuta cómo se

puede hacer esto o explique por qué el problema es inevitable.

16.3 Rendimiento de *pipeline*

Una *pipeline* de instrucción de 10 etapas corre a una tasa de reloj de 1 GHz. El esquema de adelantamiento de datos y la mezcla de instrucciones son tales que para 15% de las instrucciones se debe insertar una burbuja en la *pipeline*, para 10% dos burbujas y para 5% cuatro burbujas. La implementación de ciclo sencillo equivalente conduciría a una tasa de reloj de 150 MHz.

- a) ¿Cuál es la reducción en el rendimiento total de *pipeline* sobre la *pipeline* ideal como resultado de las burbujas?
- b) ¿Cuál es la aceleración de la implementación encauzada sobre la implementación de ciclo sencillo?

16.4 Cálculo de CPI para procesadores encauzados

Suponga que todos los riesgos de datos se pueden eliminar mediante adelantamiento o calendarización adecuada de instrucciones de máquina. Considere dos mezclas de instrucciones correspondientes a los programas “gcc” (*branch*, 19%; *jump*, 2%; otras, 79%) y

spice (Branco, 6%; *jump*, 2%; otros, 92%). Encuentre el CPI promedio para cada mezcla de instrucciones, si supone una ruta de datos encauzada de cinco etapas, probabilidad promedio de 30% de que se tomará una bifurcación y:

- Atasco total para instrucciones *branch* y *jump*
- Continuación de la ejecución si se supone que no se toma la bifurcación

Explique todas sus suposiciones. En cada caso, compare la tasa MIPS resultante si supone una tasa de reloj encauzada de 500 MHz, con una implementación de ciclo sencillo no encauzada que usa un reloj de 125 MHz.

16.5 Operación de unidad de adelantamiento de datos

Considere la unidad de adelantamiento de datos de la figura 16.4. Dada una secuencia de instrucciones, puede rastrear su progreso en la *pipeline*, determinar cuándo alguna unidad de adelantamiento se activa y causa una entrada distinta que la más alta del multiplexor asociado a enviar a la ALU. Realice el análisis anterior en la secuencia de instrucciones que se encuentra en:

- Solución al ejemplo 5.3
- Solución al ejemplo 5.4
- Solución al ejemplo 5.5
- Problema 5.11
- Solución al ejemplo 6.2
- Figura 7.2

16.6 Diseño de unidad de adelantamiento de datos

El diseño de la unidad de adelantamiento de datos depende de los detalles de la *pipeline*:

- Presente el diseño lógico completo para las unidades de adelantamiento de datos superior e inferior de la figura 16.4.
- Presente un diagrama de bloques para una unidad de adelantamiento de datos para la *pipeline* modificada de la figura 15.11. Establezca todas sus suposiciones.

16.7 Adelantamiento de datos y detección de riesgos

En la discusión del adelantamiento de datos (figura 16.4) se consideró el suministro de la versión más reciente o actual de los datos de registro sólo a la ALU.

En virtud de que los contenidos de registro también se usan en el recuadro *next address* de la figura 15.9, el adelantamiento se debe extender a dicha unidad.

- Discuta el diseño de una unidad de adelantamiento para garantizar bifurcación y salto correctos.
- ¿Es necesario también extender el detector de riesgo de datos de la figura 16.5 para este propósito?

16.8 Bifurcación retardada

Se consideró mover una instrucción que originalmente aparecía antes de la instrucción *branch* a la rendija de retardo de bifurcación. ¿Es factible llenar ésta con una instrucción que se encuentra en la dirección destino de la bifurcación? Explique. ¿Qué hay respecto de no hacer nada, de modo que la rendija de retardo de bifurcación se ocupe con la instrucción que sigue a la bifurcación y constituye la que se ejecutará si la bifurcación no se toma?

16.9 Cambio de direccionamiento en MicroMIPS

Suponga que las instrucciones *load* y *store* de MicroMIPS se redefinen para requerir que la dirección de memoria completa se proporcione en un registro. En otras palabras, las instrucciones de acceso a memoria redefinidas no contendrían *offset* y no requerirían cálculo de dirección. Describa los efectos de este cambio sobre:

- Diseño de la ruta de datos encauzada
- Rendimiento de la implementación encauzada
- Colocación y diseño de unidad(es) de adelantamiento de datos
- Diseño de la unidad de detección de riesgo de datos

16.10 Evitación de bifurcaciones

Considere añadir instrucciones de copiado adicionales al conjunto de instrucciones de MicroMIPS. La instrucción *cpyz rd, rs, rt* realiza la operación $rd \leftarrow (rs) \text{ if } (rt) = 0$. La instrucción *cpyn* es similar, excepto que se usa la condición $(rt) \neq 0$.

- Demuestre que las nuevas instrucciones permiten colocar el mayor de dos valores de registro en un tercer registro sin usar una instrucción *branch*. Puede usar *\$at* como registro temporal.
- Escriba una secuencia equivalente de instrucciones MicroMIPS para lograr la misma tarea de la parte a), pero sin usar las nuevas instrucciones.

- c) ¿Qué modificaciones se necesitan en la ruta de datos encauzada para implementar estas nuevas instrucciones?
- d) Compare el rendimiento de *pipeline* para las secuencias de instrucciones de las partes a) y b).

16.11 Predicción de bifurcación

- a) Resuelva el ejemplo 16.1 después del siguiente cambio: el *ciclo* exterior tiene una bifurcación en el final, como antes, pero el interior tiene una bifurcación de salida hacia adelante cerca del comienzo y un *jump* al final.
- b) Repita la parte a), con el tipo de bifurcaciones usadas en los *ciclos* interior y exterior invertidos (bifurcación hacia adelante en el *ciclo* exterior y hacia atrás en el interior).

16.12 Implementación de predicción de bifurcación

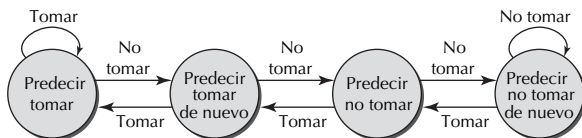
La figura 16.7 muestra el diagrama de bloques para una unidad de predicción de bifurcación basada en historia.

- a) Explique por qué se necesita el comparador.
- b) Presente el diseño del bloque lógico bajo el multiplexor, si supone 1 bit de historia.
- c) Repita la parte b) con dos bits de historia, en concordancia con el diagrama de estado de la figura 16.6.

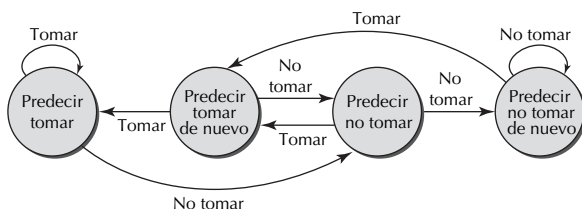
16.13 Predicción de bifurcación basado en historia

Considere el esquema de predicción de bifurcación definido por el diagrama de estado de la figura 16.6.

- a) ¿Cómo se compara el método definido por el diagrama siguiente, con el de la figura 16.6?



- b) Repita la parte a) para el siguiente esquema:



- c) Compare y contraste los esquemas de predicción de bifurcación definidos en las partes a) y b).
- d) Discuta cuál de los cuatro estados se elegiría como el estado inicial en cada esquema.

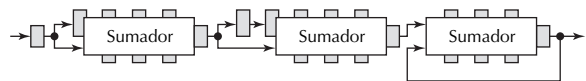
16.14 Desenrollado del ciclo

El desenrollado del *ciclo* se discutió casi al final de la sección 16.5 como una forma de reducir la penalización de bifurcación y ofrecer mayor oportunidad para la emisión múltiple de instrucciones. Para *ciclos* en cada uno de los siguientes fragmentos de programa, muestre cómo desenrollar el *ciclo* una o dos veces afecta el rendimiento de *pipeline* con una o dos instrucciones emitidas en cada ciclo (cinco casos a comparar: no desenrollar, más las cuatro combinaciones anteriores).

- a) Solución al ejemplo 5.4
- b) Solución al ejemplo 5.5
- c) Problema 5.11
- d) Figura 7.2

16.15 Pipelines de propósito especial

Considere la siguiente *pipeline* de cálculo de 15 etapas que acepta una secuencia de números en ciclos de reloj sucesivos y produce una secuencia de números en la salida. Los recuadros sombreados son registros *pipeline* y cada uno de los bloques grandes representa un sumador encauzado de cuatro etapas. Explique cómo la secuencia de salida se relaciona con las entradas.



16.16 Rendimiento de *pipelining* y su límite

Se diseñará una ruta de datos lineal que consta de partes con las siguientes latencias, en orden de izquierda a derecha: 0.2 ns, 0.6 ns, 0.5 ns, 0.6 ns, 0.3 ns, 0.8 ns. Estas partes son indivisibles, de modo que cualquier *latch* encauzado se debe insertar entre componentes consecutivos. La inserción de *latches* introduce una cabecera de etapa de 0.2 ns. Determine la latencia y rendimiento total de la ruta de datos sin encauzamiento (sólo un conjunto de *latches* en la salida) y con 2, 3, 4,... etapas en la *pipeline*, que continúe hasta que se alcance el rendimiento total máximo.

16.17 Excepciones debidas a faltas o errores

En algunas computadoras, cuando una instrucción no se ejecuta adecuadamente o se sospecha que sus resultados son incorrectos, la unidad de control intenta ejecutar de nuevo la instrucción, para el caso de que el problema se deba a una falta transitoria en hardware

o alguna rara combinación de condiciones de operación. Esto último se denomina *reintento de instrucción*. Si supone que existe hardware especial que señala la necesidad de reintento de instrucción, discuta cómo se puede implementar esta característica en procesadores no encauzados y encauzados.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Crag96] Cragon, H. G., *Memory Systems and Pipelined Processors*, Jones and Bartlett, 1996.
- [Dube91] Dubey, P. y M. Flynn, "Branch Strategies: Modeling and Optimization", *IEEE Trans. Computers*, vol. 40, núm. 10, pp. 1159-1167, octubre de 1991.
- [Flyn95] Flynn, M. J., *Computer Architecture: Pipelined and Parallel Processor Design*, Jones and Bartlett, 1995.
- [Hint01] Hinton, G., *et al.*, "The Microarchitecture of the Pentium 4 Processor", *Intel Technology J.*, 12 pp., primer trimestre, 2001.
- [Lilj88] Lilja, D., "Reducing the Branch Penalty in Pipelined Processors", *IEEE Computer*, vol. 21, núm. 7, pp. 47-55, julio de 1988.
- [Smit81] Smith, J. E., "A Study of Branch Prediction Strategies", *Proceedings of the Eighth International Symposium on Computer Architecture*, pp. 135-148, mayo de 1981.
- [Smit88] Smith, J. E. y A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors", *IEEE Trans. Computers*, vol. 37, núm. 5, pp. 562-573, mayo de 1988.
- [Yeh92] Yeh, T. Y. y Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction", *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 124-134, mayo de 1992.

PARTE CINCO

DISEÑO DE SISTEMAS DE MEMORIA

“Siempre diseño algo considerándolo en su siguiente contexto más grande: una silla en una habitación, una habitación en una casa, una casa en un ambiente, un ambiente en una ciudad.”

Eliel Saarinen

“De hecho, la fantasía es un modo de memoria emancipada del orden del tiempo y el espacio.”

Samuel Taylor Coleridge

TEMAS DE ESTA PARTE

- 17. Conceptos de memoria principal
- 18. Organización de memoria caché
- 19. Conceptos de memoria masiva
- 20. Memoria virtual y paginación

El sistema de memoria de una computadora es tan importante como el CPU para determinar su rendimiento y utilidad. En la parte cuatro se usaron componentes de memoria que eran tan rápidos como la ALU y los otros elementos de ruta de datos. Aunque, de hecho, tales memorias se pueden construir, su tamaño está muy limitado por restricciones tecnológicas y económicas. La paradoja en el diseño de sistemas de memoria es que se quiere una gran cantidad de memoria para alojar programas complejos, así como conjuntos de datos extensos, y que esta gran memoria sea muy rápida como para no frenar el flujo de datos a través de la ruta de datos. El aspecto principal de esta parte del libro es mostrar cómo se resuelve esta paradoja mediante métodos arquitectónicos.

Después de revisar tecnologías y estructuras organizativas para la memoria principal de una computadora en el capítulo 17, se concluye que la tecnología actual es incapaz de proporcionar una memoria de un solo nivel que sea tan rápida como grande para las necesidades propias. En el capítulo 18 se muestra cómo el uso de uno o dos niveles de memoria caché ayudan a resolver la brecha de rapidez entre el CPU y la gran memoria principal. En virtud de que incluso una gran memoria principal todavía no es tan grande para alojar todas las necesidades de almacenamiento, en el capítulo 19 se estudian tecnologías y organizaciones para memorias secundarias o masivas. El capítulo 20, que trata la memoria virtual, explica cómo, al automatizar las transferencias de datos entre las memorias primaria y secundaria, se puede ofrecer la ilusión de una memoria principal multigigabyte a bajo costo.

CONCEPTOS DE MEMORIA PRINCIPAL

“Es una mala suerte de memoria que sólo funcione hacia atrás.”

Lewis Carroll

“La existencia del olvido nunca se ha probado; sólo se sabe que algunas cosas no llegan a la mente cuando se les quiere.”

Friedrich W. Nietzsche

TEMAS DEL CAPÍTULO

- 17.1** Estructura de memoria y SRAM
- 17.2** DRAM y ciclos de regeneración
- 17.3** Impactar la pared de memoria
- 17.4** Memorias encauzada e interpolada
- 17.5** Memoria no volátil
- 17.6** Necesidad de una jerarquía de memoria

La tecnología de memoria principal ahora en uso es muy diferente de los tambores magnéticos y memorias centrales de las primeras computadoras digitales. Las memorias de semiconductores actuales son al mismo tiempo más rápidas, más densas y más baratas que sus antecesoras. Este capítulo se dedica a revisar la organización de memoria, incluidas las tecnologías SRAM y DRAM que usualmente se usan para memorias caché rápidas y memoria principal más lenta, respectivamente. Se muestra que la memoria ya se ha convertido en un factor muy limitante en el rendimiento de las computadoras y se discute cómo ciertas técnicas organizativas, como el interpolado (*interleaving*) y el encauzamiento (*pipelining*), pueden mitigar algunos de los problemas. Se concluye con la justificación de la necesidad de una jerarquía de memoria, como preparación para las discusiones acerca de las memorias caché y virtual.

■ 17.1 Estructura de memoria y SRAM

La memoria de acceso aleatorio estática (SRAM, por sus siglas en inglés) constituye un gran arreglo de celdas de almacenamiento a las que se accede como registros. Una celda de memoria SRAM usualmente requiere entre cuatro y seis transistores por bit y retiene los datos almacenados en tanto esté encendida. Esta situación está en contraste con la memoria de acceso aleatorio dinámica (DRAM, por sus siglas en inglés), a discutir en la sección 17.2, que usa sólo un transistor por bit y se debe regenerar

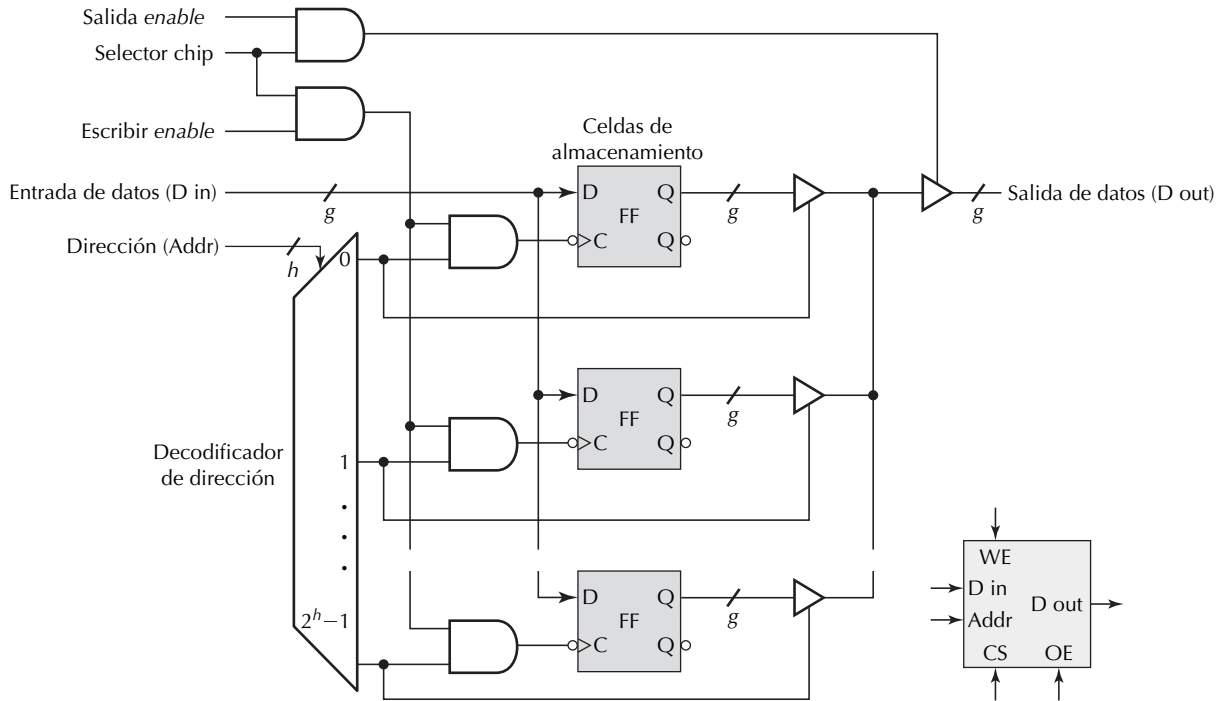


Figura 17.1 Estructura interna conceptual de un chip SRAM $2^h \times g$ y su representación abreviada.

periódicamente para evitar pérdida de datos almacenados. Tanto SRAM como DRAM perderían datos si se desconectan de la potencia; de ahí su designación de memoria *volátil*. Como se muestra en la figura 2.10, un chip SRAM $2^h \times g$ tiene una entrada de h bits que porta la dirección. Esta dirección se decodifica y usa para seleccionar una de las 2^h ubicaciones, indexadas de 0 a $2^h - 1$, cada una de las cuales tiene g bits de ancho. Durante una operación de escritura, los g bits de datos se proporcionan al chip al copiar en la ubicación direccionada. Para una operación de lectura, g bits de datos se leen de la celda direccionada. La señal *Chip select* (selección de chip, CS) se debe postular para operaciones de lectura y escritura, mientras que *Write enable* (habilitar escritura, WE) se postula para escritura y *Output enable* (habilitar salida, OE) para lectura (figura 17.1). Un chip que no se selecciona no realiza operación alguna, sin importar los valores de sus otras señales de entrada.

Para facilitar la comprensión, las celdas de almacenamiento de la figura 17.1 se bosquejan como *flip-flops* D activados por flanco. En la práctica se emplean *latches* D porque usar *flip-flops* conduciría a complejidad adicional en las celdas y, por tanto, menos celdas en un chip. El uso de *latches* no causa mayor problema durante las operaciones de lectura; sin embargo, para operaciones de escritura, se deben satisfacer requisitos de temporización más rigurosos con el fin de que garanticen que los datos se escriben adecuadamente en la localidad deseada y sólo en dicha localidad. La discusión detallada de tales consideraciones de temporización no se tratan en este libro [Wake01].

Una SRAM síncrona (SSRAM) proporciona una interfaz más limpia al diseñador mientras todavía usa *latches* D internamente. El comportamiento síncrono se logra al hacer que los registros internos retengan direcciones, datos y entradas de control (y acaso la salida de datos). Puesto que las señales de entrada se proporcionan en un ciclo de reloj, mientras que a la memoria se accede en el ciclo siguiente, no hay peligro de que las entradas al arreglo de memoria cambien en tiempos inoportunos.

Si el tamaño de un chip SRAM $2^h \times g$ es inadecuado para las necesidades de almacenamiento, se pueden usar múltiples chips. Se usan k/g chips en paralelo para obtener palabras de datos de k bits. Se usan $2^m/2^h = 2^{m-h}$ hileras de chips para obtener una capacidad de 2^m palabras.

Ejemplo 17.1: SRAM de múltiples chips Muestre cómo se pueden usar chips SRAM de $128\text{K} \times 8$ para construir una unidad de memoria de $256\text{K} \times 32$.

Solución: Como consecuencia de que el ancho de palabra deseado es cuatro veces el del chip SRAM, se deben usar cuatro chips en paralelo. Se necesitan dos hileras de chips para duplicar el número de palabras de 128K a 256K . La estructura requerida se muestra en la figura 17.2. El bit más significativo de la dirección de 18 bits se usa para elegir la hilera 0 o la hilera 1 de los chips SRAM. Todas las otras señales de control de chip se ligan externamente a las señales de control de entrada comunes de la unidad de memoria.

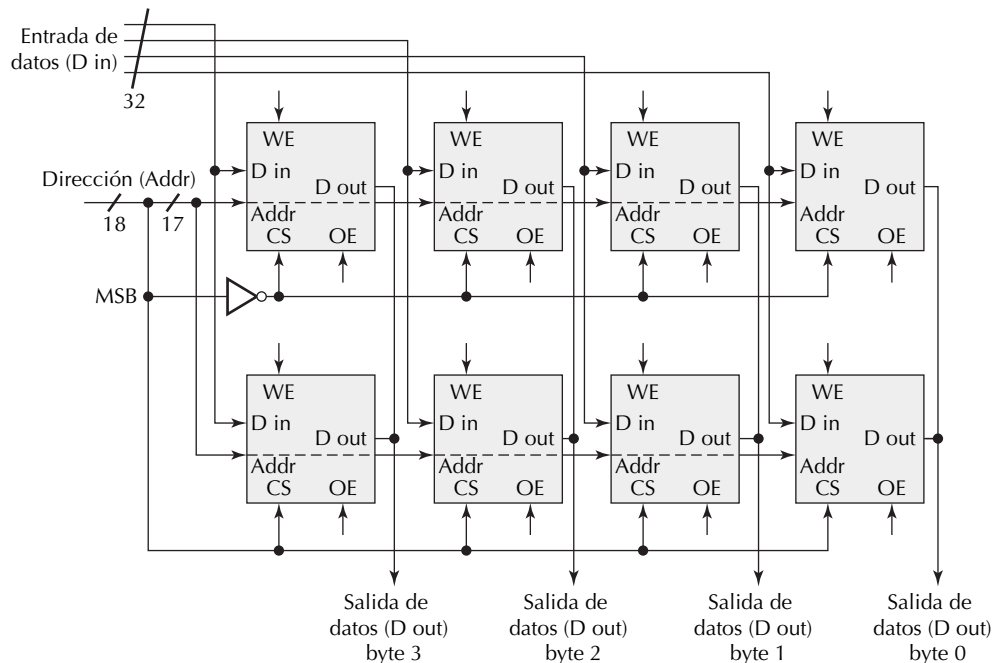


Figura 17.2 Ocho chips SRAM de $128\text{K} \times 8$ que forman una unidad de memoria de $256\text{K} \times 32$.

Con frecuencia, los pines de entrada y salida de datos de un chip SRAM se comparten, los pines de entrada y salida no compartidos se conectan al mismo bus de datos bidireccional. Esto último tiene sentido, pues, a diferencia de un archivo de registro que se lee y escribe durante el mismo ciclo de reloj, una unidad de memoria realiza una operación de lectura o escritura, pero no ambas al mismo tiempo. Con el propósito de garantizar que los datos de salida de la SRAM no interfieren con los datos de entrada proporcionados durante una operación de escritura, se debe modificar ligeramente el circuito de control que se muestra en la parte superior de la figura 17.1. Esta modificación, que conlleva deshabilitar la salida de datos cuando se postula *Write enable*, se muestra en la figura 17.3.

Los primeros chips de memoria estaban organizados como memorias $2^h \times 1$. Esto último sucede porque las capacidades de los chips eran muy limitadas y cualquier aumento de memoria requería de

Figura 17.3 Cuando la entrada y salida de datos de un chip SRAM se comparten o conectan a un bus de datos bidireccional, la salida se debe deshabilitar durante las operaciones de escritura.

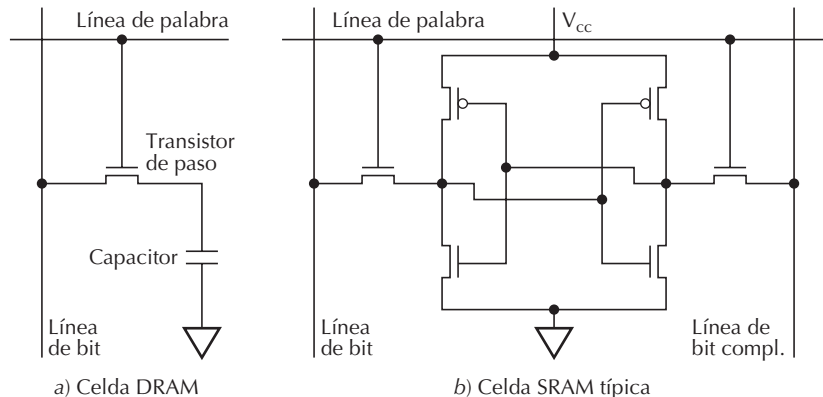
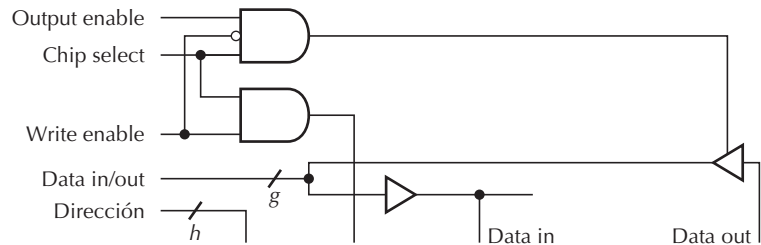


Figura 17.4 La celda DRAM de transistor sencillo, que es más simple que la celda SRAM, conduce a chips de memoria DRAM densos de alta capacidad.

muchos chips RAM. Tener un bit de cada palabra de datos en un chip ofrece la ventaja de aislamiento de error; un chip que falle afectará a no más de un bit en cada palabra; por tanto, permitirá códigos para detectar y corregir errores para señalar o enmascarar efectivamente el error. En la actualidad no es raro que toda la memoria necesite una aplicación para encajar en uno o un puñado de chips. Por tanto, los chips con palabras con tamaño de byte se han vuelto muy comunes. Sin embargo, más allá de entrada y salida de datos de ocho bits, la limitación de pin hace poco atractivo ensanchar aún más las palabras de memoria. Observe que, como se muestra en la figura 2.10, se construye un arreglo de memoria de acceso aleatorio con palabras muy anchas que se leen en un *buffer* interno, donde la porción correspondiente a una palabra de memoria externa se selecciona para salida. Así, aparte de la limitación de pin, no hay impedimento real para tener palabras de memoria más anchas.

■ 17.2 DRAM y ciclos de regeneración

Es imposible construir un elemento biestable sólo con un transistor. Para permitir celdas de memoria de transistor sencillo, que conducen a la mayor densidad de almacenamiento posible en un chip, y a costo por bit muy bajo, la memoria de acceso aleatorio dinámica (DRAM) almacena datos como carga eléctrica en pequeños capacitores, a los que se accede mediante un transistor de paso MOS. La figura 17.4 muestra tal celda en forma esquemática. Cuando la línea de palabra se postula, el voltaje bajo (alto) en la línea de bit causa que el capacitor se descargue (cargue) y, por tanto, almacena 0 (1). Para leer una celda DRAM, primero se *precarga* la línea de bit a medio voltaje. Luego este voltaje se jala ligeramente abajo o arriba hasta la postulación de la línea de palabra, dependiendo de si la celda

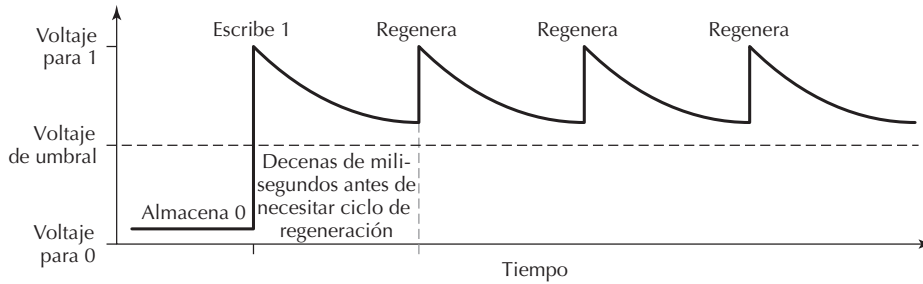


Figura 17.5 Variaciones en el voltaje a través del capacitor de una celda DRAM después de escribir un 1 y subsecuentes operaciones de regeneración.

almacena 0 o 1. Este cambio de voltaje lo detecta un amplificador, que recupera un 0 o un 1 en concordancia. Puesto que el acto de leer destruye el contenido de la celda, tal *lectura destructiva* de datos se debe seguir inmediatamente con una operación de escritura para restaurar los valores originales.

La fuga de carga del pequeño capacitor que se muestra en la figura 17.4 causa que los datos se borren después de una fracción de segundo. En consecuencia, las DRAM se deben equipar con circuitos especiales que periódicamente regeneren los contenidos de memoria. La regeneración sucede para todas las hileras de memoria mediante la lectura de cada hilera y su reescritura para restaurar la carga al valor original. La figura 17.5 muestra cómo la fuga conduce al decaimiento de carga en una celda DRAM que almacena 1, y cómo la regeneración periódica restaura la carga justo antes de que el voltaje a través del capacitor caiga por abajo de cierto umbral. Las DRAM son mucho más baratas, pero también más lentas, que las SRAM. Además, cierto ancho de banda de memoria potencial en DRAM se pierde en las operaciones de reescritura para restaurar los datos destruidos durante lectura y para ciclos de regeneración.

Ejemplo 17.2: Pérdida de ancho de banda para ciclos de regeneración Un chip DRAM de 256 Mb está organizado externamente como una memoria de $32\text{M} \times 8$ e internamente como un arreglo cuadrado de $16\text{K} \times 16\text{K}$. Cada hilera se debe regenerar al menos una vez cada 50 ms con el fin de prever pérdida de datos; regenerar una hilera toma 100 ns. ¿Qué fracción del ancho de banda de memoria total se pierde en ciclos de regeneración?

Solución: Regenerar las 16K hileras toma $16 \times 1024 \times 100 \text{ ns} = 1.64 \text{ ms}$. Por tanto, de cada periodo de 50 ms se pierden 1.64 ms en ciclos de regeneración. Esto último representa $1.64/50 = 3.3\%$ del ancho de banda total.

En parte debido a la gran capacidad de los chips DRAM, que requeriría un gran número de pines de direcciones, y en parte porque, dentro del chip, el proceso de lectura de cualquier forma ocurre en dos pasos (acceso a hilera, selección de columna), DRAM usualmente tiene la mitad de pines de dirección que los dictados por su capacidad de palabra. Por ejemplo, un chip DRAM $2^{18} \times 4$ puede tener sólo nueve pines de direcciones. Para ingresar una palabra, primero se proporciona su dirección de hilera dentro del arreglo de memoria, junto con una señal de *habilitar (strobe) dirección de hilera* (*row address strobe*, RAS) que indica la disponibilidad de la dirección de hilera a la unidad de memoria, que lo copia en un registro interno. Pero después se proporciona la dirección de columna mediante las mismas líneas de dirección, concurrentemente con la señal *habilitar (strobe) dirección de columna* (*column address strobe*, CAS). La figura 17.6 muestra un paquete DRAM de 24 pines típico con 11 pines de dirección y cuatro de datos.

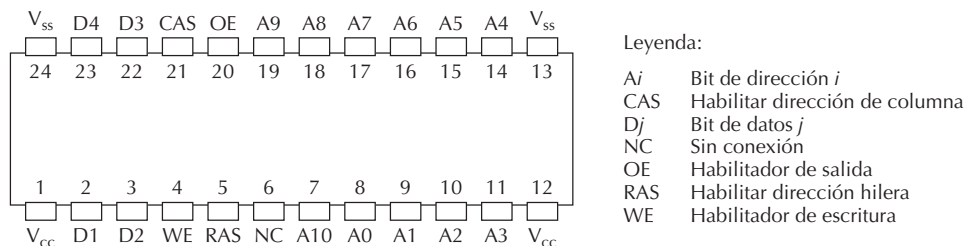


Figura 17.6 Paquete DRAM típico que alberga una memoria $16M \times 4$.

A continuación se describe la temporización de eventos en un chip DRAM durante las tres operaciones de regeneración, lectura y escritura. La regeneración se puede realizar al proporcionar una dirección de hilera al chip. En este modo de regeneración de sólo RAS, el flanco de subida de RAS causa que se lea la hilera correspondiente en un *buffer* de hilera interno y el flanco de bajada causa que se reescriba el contenido de aquél. Las operaciones son asíncronas y no se involucra reloj. Por esta razón, la temporización de la señal RAS es bastante crucial. Un ciclo de lectura comienza como ciclo de regeneración, pero la señal CAS se usa para habilitar los bits de salida del chip, que se seleccionan del contenido del *buffer* de hilera mediante la dirección de columna proporcionada concurrentemente con CAS. Un ciclo de escritura también comienza como un ciclo de regeneración. Sin embargo, antes de que el flanco de subida de RAS señale una operación de reescritura, los datos en el *buffer* de hilera se modifican al postular las señales *Write enable* y CAS con la aplicación de la dirección de columna. Esto causa que los datos en la parte adecuada del *buffer* de hilera se modifiquen antes de que todo el *buffer* se reescriba.

Las DRAM típicas tienen otros modos de operación además de regeneración, lectura y escritura sólo RAS. Una de las más útiles de dichas variaciones se conoce como *modo de página*. En éste, al seleccionarse una hilera y leerse en el *buffer* de hilera interno, los accesos subsecuentes a palabras en la misma hilera no requieren el ciclo completo de memoria. Cada acceso subsecuente a una palabra en la misma hilera se logra al suministrar la correspondiente dirección de columna, por tanto, es significativamente más rápido.

Es evidente, a partir de lo visto hasta el momento, que la temporización de señales es bastante crucial para la operación DRAM. Esto no sólo hace que el proceso de diseño sea difícil, también conduce a menor rendimiento debido a la necesidad de ofrecer márgenes de seguridad adecuados. Por esta razón, las variaciones de DRAM síncrona (SDRAM) se han vuelto muy populares. Como fue el caso para SSRAM, la operación interna de la SDRAM sigue asíncrona. Sin embargo, la interfaz externa es síncrona. Usualmente, la dirección de hilera se proporciona en un ciclo de reloj y la dirección de columna en el siguiente. También en el primer ciclo se proporciona una palabra de comando que especifica la operación a realizar. El acceso real a los contenidos de memoria toma varios ciclos de reloj. No obstante, la memoria está organizada internamente en múltiples bancos, de modo que, cuando se inicia el acceso a un banco, el tiempo de espera se puede usar para procesar más comandos de entrada que pueden involucrar acceso a otros bancos. De esta forma se puede soportar un rendimiento total mucho mayor. Esto es particularmente cierto porque a las DRAM modernas se accede en estallidos para leer secuencias de instrucciones o bloques multipalabra de caché. Con frecuencia se permite una *longitud de estallido* (*burst length*) como parte del comando de entrada al controlador DRAM, que entonces causa la transferencia del número especificado de palabras, una por ciclo de reloj, cuando la hilera se lee en el *buffer* de hilera.

Actualmente son de amplio uso dos implementaciones mejoradas de SDRAM. La primera se conoce como DRAM con tasa de datos doble (DDR-DRAM), que duplica la tasa de transferencia de la DRAM al usar ambos flancos de la señal de reloj para señalar acciones. Otra, Rambus DRAM

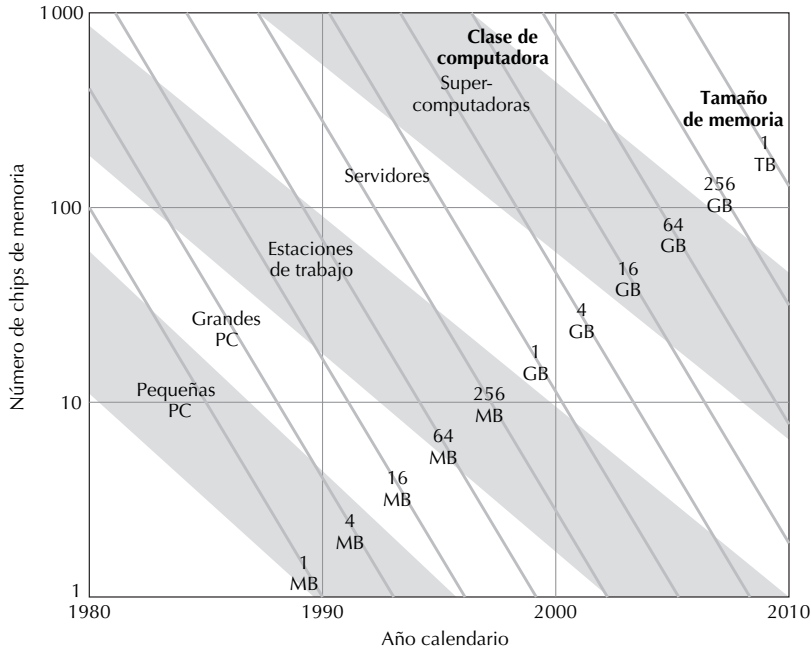


Figura 17.7 Tendencias en memoria principal DRAM. Extrapolada de [Przy94].

(RDRAM), combina la operación de flanco doble con un canal muy rápido, pero relativamente estrecho, entre el controlador DRAM y el arreglo de memoria. La estrechez del canal tiene la intención de facilitar la producción de un vínculo libre de sesgo y de alta calidad que se puede conducir a tasas de reloj muy elevadas. Se espera que las RDRAM se adhieran a las especificaciones de canal Rambus para temporización y asignación de pines (*pinout*) [Shri98], [Cupp01].

La figura 17.7 presenta una interesante vista del progreso de las densidades de DRAM y capacidades de chip desde 1980. Se muestra, por ejemplo, que una memoria principal de 512 MB, que habría requerido cientos de chips DRAM en 1990, se podía construir con un puñado de chips en 2000 y ahora requiere sólo un chip DRAM. Más allá de esto, la misma unidad de memoria se puede integrar en un chip VLSI con un CPU y otros dispositivos requeridos para formar una poderosa computadora de un solo chip.

17.3 Impactar la pared de memoria

Mientras que tanto las densidades de SRAM y DRAM como las capacidades de chip aumentaron desde la invención de los circuitos integrados, las mejoras en rapidez no han sido tan impresionantes. De este modo, los procesadores cada vez más rápidos y la mayor rapidez de reloj han conducido a un ensanchamiento de la brecha entre rendimiento de CPU y ancho de banda de memoria. En otras palabras, el rendimiento y la densidad del procesador, así como la densidad de la memoria, aumentan exponencialmente según predice la ley de Moore. Esto último lo representa la pendiente superior en la gráfica semi-logarítmica de la figura 17.8. Sin embargo, el rendimiento de memoria ha mejorado con una pendiente mucho más modesta. Esta brecha de rendimiento CPU-memoria es particularmente problemática para DRAM. La rapidez de las SRAM han mostrado mayor mejora, no sólo para chips de memoria sino también porque las memorias caché, que constituyen una aplicación primaria de las SRAM, ahora se integran rutinariamente con el procesador en el mismo chip, ello reduce los tiempos de propagación de señal y otras cabeceras de acceso involucradas cuando se debe acceder a una unidad fuera del chip.

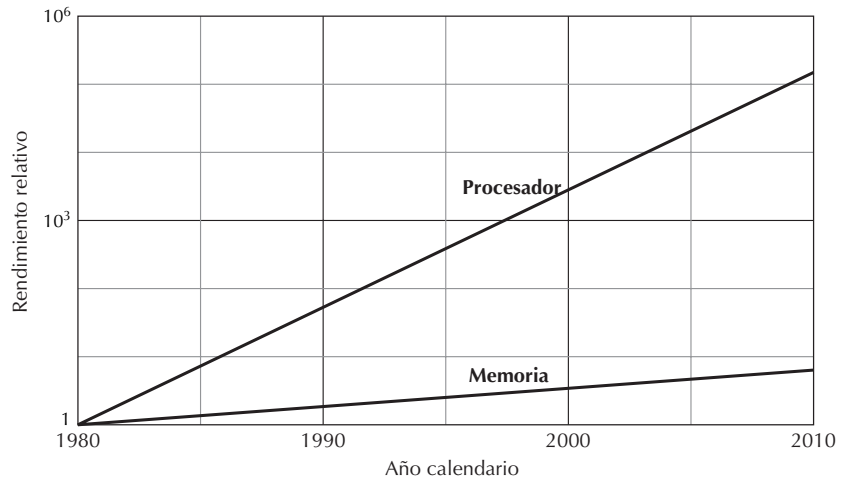


Figura 17.8 La densidad y capacidad de memoria crecieron junto con la potencia y complejidad de CPU, pero la rapidez de la memoria no ha seguido el paso.

Por estas razones, la latencia de memoria se ha convertido en un serio problema en las computadoras modernas. Se argumenta que, incluso ahora, mayores mejoras en la rapidez de procesador y tasa de reloj no significan ganancia notable en el rendimiento al nivel aplicación porque el rendimiento percibido por el usuario está completamente limitado al tiempo de acceso a memoria. Esta condición se denomina “impactar la pared de memoria”, que señala a esta última como una barrera para mayor progreso [Wulf95]. Por ejemplo, se nota que, si continúan las tendencias que se muestran en la figura 17.8, hacia 2010 cada acceso de memoria tendrá una latencia promedio equivalente a cientos de ciclos de procesador. Por tanto, incluso si sólo 1% de las instrucciones requieren acceso a memoria principal DRAM, es la latencia DRAM, no la rapidez del procesador, la que indica el tiempo de ejecución del programa.

Estos argumentos son muy desalentadores, pero no toda esperanza está perdida. Incluso si un gran descubrimiento no resolviera el problema de la brecha de rendimiento, se podrían usar muchos de los métodos existentes y propuestos para mover la pared de memoria.

Una forma de superar la brecha entre la rapidez del procesador y la de la memoria, consiste en usar palabras muy anchas de modo que cada acceso a la memoria lenta recupere gran cantidad de datos; entonces se usa un mux para seleccionar la palabra adecuada a adelantar al procesador (figura 17.9). En este contexto, las palabras restantes que se leen (*fetch*) pueden ser útiles para instrucciones subsecuentes. Más tarde se verá que los datos se transfieren entre memoria principal y caché en palabras múltiples conocidas como línea de caché. Por tanto, tales accesos de palabra ancha son ideales para los sistemas modernos con caché. Otros dos enfoques, a saber, las memorias interpoladas (*interleaved*) y encauzadas (*pipelined*), tienen un significado especial y necesitan descripciones más detalladas; en consecuencia, se discuten por separado en la sección 17.4.

Incluso si se supone que la creciente brecha de rendimiento no se puede superar a través de los métodos arquitectónicos ya mencionados, la única conclusión que se puede extraer es que el rendimiento de un solo procesador estará limitado por la latencia de memoria. En teoría, se puede aplicar la potencia de decenas, acaso incluso de cientos, de procesadores a la solución del problema. Aunque cada uno de estos procesadores puede estar limitado por el problema de la memoria, colectivamente pueden proporcionar un nivel de rendimiento que se pueda ver como pasando justo a través de la pared de memoria. Los métodos de procesamiento en paralelo se estudiarán en la parte siete.

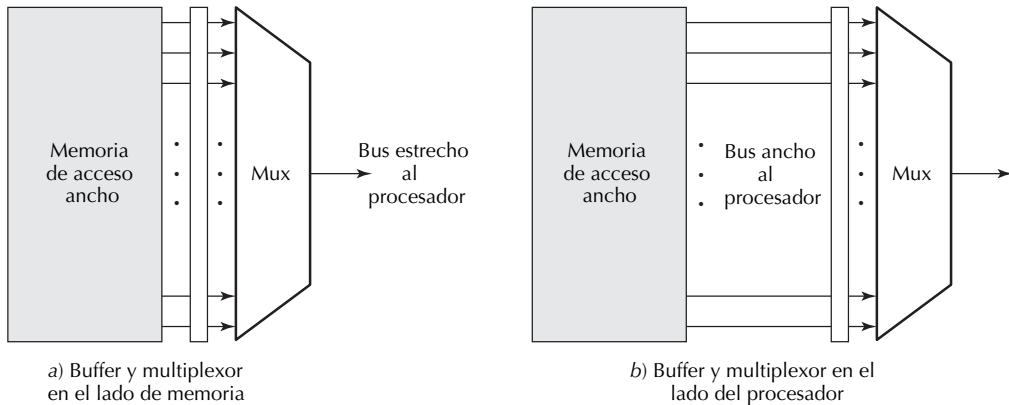


Figura 17.9 Dos formas de usar una memoria de acceso ancho para salvar la brecha de rapidez entre procesador y memoria.

17.4 Memorias encauzada e interpolada

Las unidades de memoria simple permiten el acceso secuencial a sus localidades (una a la vez). Aunque a los archivos de registro se les proporcionan puertos múltiples para permitir muchos accesos simultáneos, este enfoque para elevar el ancho de banda sería muy costoso para las memorias. ¿Entonces cómo se puede aumentar el rendimiento total de memoria de modo que se pueda acceder a más datos por unidad de tiempo? Los dos esquemas principales para este propósito son los accesos de datos encauzados y paralelos.

El encauzamiento (*pipelining*) de memoria permite que la latencia de acceso de memoria se disperse sobre muchas etapas de *pipeline*; por ende, aumenta el rendimiento total de la memoria. Sin *pipelining* en los accesos de instrucción y datos en la ruta de datos de un procesador, el rendimiento estará limitado por las etapas de *pipeline* que contengan referencias de memoria. El *pipelining* de memoria es posible porque el acceso a un banco de memoria consta de muchos eventos o pasos secuenciales: decodificación de dirección de hilera, lectura de hilera desde la matriz de memoria, escritura en una palabra o adelantarla a la salida (con base en la dirección de columna) y reescritura del *buffer* de hilera modificado en la matriz de memoria, si fuera necesario. Al separar algunos de estos pasos y visualizarlos como diferentes etapas de una *pipeline*, se puede iniciar un nuevo acceso a memoria cuando el anterior se mueve a la segunda etapa de *pipeline*. Como siempre, el *pipelining* puede aumentar un poco la latencia, pero mejora el rendimiento total al permitir que varios accesos de memoria “vuelen” mediante las etapas de *pipeline* de memoria.

Además del acceso a la memoria física, como ya se discutió, el acceso a memoria involucra otras operaciones de soporte que formen etapas adicionales en la *pipeline* de memoria. Éstas incluyen posibles cambios de dirección antes de ingresar a la memoria (caché o principal) y comparación de etiquetas (*tags*) después de ingresar a una memoria caché para garantizar que la palabra de datos leída del caché es la solicitada (en este punto no queda claro por qué pueden ser diferentes ambas). En el capítulo 18 se estudiarán los conceptos relacionados con el acceso a memoria caché, mientras que en el capítulo 20 se discutirá el cambio de dirección para memoria virtual.

La figura 17.10 muestra una memoria caché encauzada con base en las nociones discutidas antes. Existen cuatro etapas en la *pipeline*: una etapa de cambio de dirección, dos etapas para el acceso a la memoria real y una etapa para comparación de etiqueta (*tag*) para garantizar la validez de los datos leídos (*fetch*). La etapa de cambio de dirección no siempre es necesaria. Por un lado, los cachés no

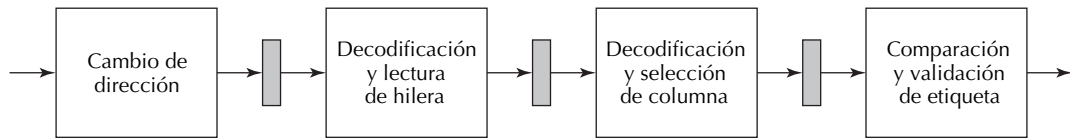


Figura 17.10 Memoria caché encauzada.

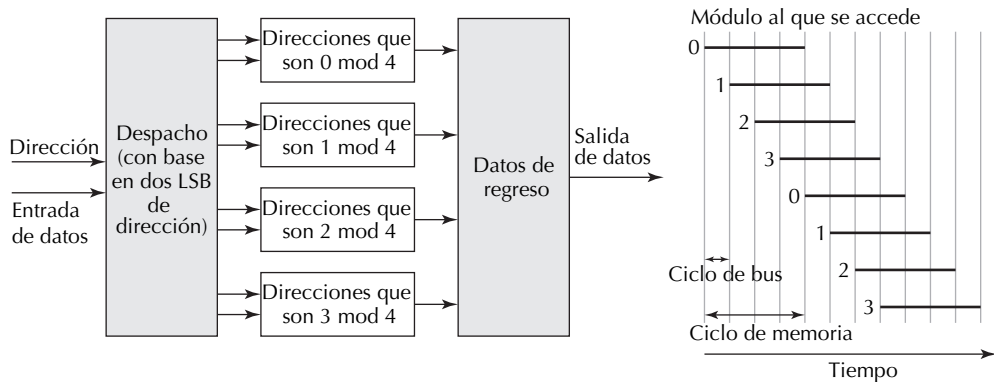


Figura 17.11 La memoria interpolada es más flexible que la memoria de acceso ancho en cuanto a que puede manipular múltiples accesos independientes a la vez.

siempre lo requieren; por otro, aquél a veces se puede traslapar con el acceso a memoria física (estos puntos se estudian en el capítulo 18). Para el caso de una caché de instrucción, la etapa de comparación de etiqueta se puede traslapar con la decodificación de instrucciones y el acceso a registro, de modo que su latencia se vuelve transparente, mientras que para caché de datos, que usualmente se sigue por reescritura de registro, no es posible el traslapamiento. Por ende, si la memoria encauzada se usara en la ruta de datos de la figura 15.11, se tendría que incluir una etapa de *pipeline* adicional entre el acceso a caché de datos y reescritura de registro.

La *memoria interpolada* permite que ocurran múltiples accesos en paralelo, pero deben estar en direcciones ubicadas en diferentes bancos de memoria. La figura 17.11 muestra el diagrama de bloques para una unidad de memoria *interpolada de cuatro vías*. Cada uno de los cuatro bancos de memoria retiene direcciones que tienen el mismo residuo cuando se dividen entre 4; por tanto, el banco 0 tiene direcciones que son de la forma $4j$, el banco 1 tiene direcciones $4j + 1$, etcétera. Conforme llega cada solicitud de acceso a memoria, se envía al banco de memoria apropiado con base en los dos LSB en la dirección. En cada ciclo de reloj se puede aceptar una nueva solicitud, aun cuando un banco de memoria tome muchos ciclos de reloj para proporcionar los datos solicitados o escribir una palabra. En virtud de que todos los bancos tienen la misma latencia, las palabras de datos solicitados surgen del lado de salida en orden de llegada.

Bajo condiciones ideales, una memoria interpolada de m vías logra un ancho de banda que es m veces el de un solo banco de memoria. Eso ocurre, por ejemplo, cuando se accede secuencialmente a suficientes direcciones de memoria para permitir que cada banco complete un acceso antes de que llegue el siguiente acceso al mismo banco. En el otro extremo, si todos los accesos son al mismo banco, la interpolación no ofrece beneficio de rendimiento alguno. Aun cuando las direcciones sean aleatorias, el beneficio de rendimiento es muy limitado si los accesos se deben adelantar a bancos en el orden recibido para asegurar que las palabras de salida no aparezcan fuera de orden. Esto último se entiende mejor en términos del problema de cumpleaños común en la probabilidad elemental: aun

cuando existen 365 días en un año, sólo toma 50 personas en una habitación para hacer muy probable (probabilidad de más de 95%) que al menos dos individuos tengan un cumpleaños común. El análisis estadístico muestra que, con accesos aleatorios y procesamiento en orden de solicitudes de acceso, la interpolación de m vías conduce a una mejora en el rendimiento total por un factor de \sqrt{m} , que es mucho menor que el factor ideal m . Sin embargo, las direcciones de memoria prácticas no se distribuyen aleatoriamente, sino que muestran un patrón secuencial definido, de modo que se puede esperar un rendimiento mucho mejor.

Una unidad de memoria interpolada puede formar el núcleo de una implementación de memoria encauzada. Considere la memoria interpolada de cuatro vías de la figura 17.11 como una unidad de memoria encauzada de cuatro etapas. La figura 17.11 muestra que, en tanto los accesos al mismo banco de memoria estén separados por cuatro ciclos de reloj, la *pipeline* de memoria opera sin problema. Sin embargo, dentro de la ruta de datos de un procesador, tal espaciamiento de accesos no puede garantizarse, y se necesita un mecanismo especial de detección de conflicto para garantizar la operación correcta mediante atascos cuando fuere necesario. Es muy fácil implementar tal mecanismo. Una solicitud de acceso entrante se caracteriza por un número de banco de memoria de dos bits. Este número de banco se debe verificar contra los números de banco de hasta tres accesos que todavía puedan estar en progreso (aquellos que pudieron haber comenzado en cada uno de los tres ciclos de reloj precedentes). Si alguna de estas tres comparaciones conduce a un emparejamiento, se debe atascar la *pipeline* para no permitir el nuevo acceso. De otro modo, la solicitud se acepta y envía al banco apropiado, mientras su número de banco se corre en un registro de corrimiento de cuatro posiciones para compararlo con las solicitudes entrantes en los tres ciclos siguientes.

El esquema de encauzamiento ya discutido en esta sección puede acomodar un número limitado de etapas. La memoria encauzada basada en interpolado se puede usar conceptualmente con un número arbitrario de etapas. Sin embargo, excepto en aplicaciones con patrones de acceso a memoria bastante regulares, la complejidad de los circuitos de control y la penalización por atasco ponen una cota superior práctica sobre el número de etapas. Por esta razón, el interpolado a gran escala sólo se usa en supercomputadoras vectoriales, que están optimizadas para operar sobre vectores largos cuya plantilla en memoria asegure que los accesos a memoria temporalmente cercanos se enrutan a bancos diferentes (capítulo 26).

17.5 Memoria no volátil

Tanto SRAM como DRAM requieren energía para mantener intactos los datos almacenados. Este tipo de memoria *volátil* se debe complementar con memoria *no volátil* o *estable* si los datos y programas no se pierden cuando la energía se interrumpe. Para la mayoría de las computadoras, esta memoria estable consta de dos partes: una *memoria de sólo lectura* (ROM) relativamente pequeña para retener programas de sistema cruciales que se necesitan para arrancar la máquina y un *disco duro* que conserve datos almacenados virtualmente de manera indefinida sin requerir energía. El bajo costo y la gran densidad de almacenamiento de las memorias de disco modernas las hacen ideales dispositivos de memoria estable. Debido a sus grandes capacidades, las memorias de disco juegan los papeles duales de almacenamiento estable y *almacenamiento masivo* (capítulo 19).

En los pequeños dispositivos portátiles con capacidad de espacio y batería limitadas, el tamaño y los requisitos de energía de las memorias de disco son una seria desventaja, como lo son sus partes mecánicas móviles. Por estas razones, en tales sistemas se prefiere usar dispositivos de memoria de semiconductor no volátil. Las opciones disponibles abarcan un amplio rango, desde los antiguos dispositivos de memoria de sólo lectura bien establecidos hasta los dispositivos lectura-escritura más recientes ejemplificados por las memorias tipo *flash*.

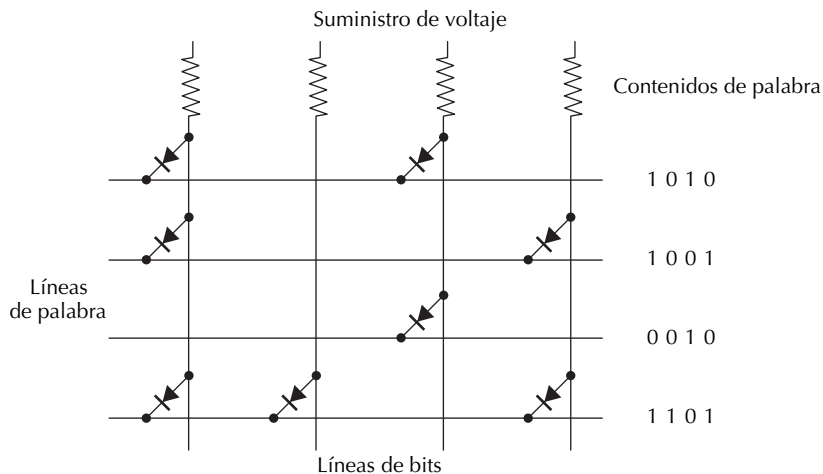


Figura 17.12 Organización de memoria de sólo lectura, donde los contenidos fijos se muestran a la derecha.

Las memorias de sólo lectura se construyen en una variedad de formas, pero todas ellas comparten la propiedad de poseer un patrón específico de 0 y 1 alambrados al momento de fabricación. Por ejemplo, la figura 17.12 muestra un segmento de cuatro palabras de una ROM en la que una línea de bit normalmente alta supone voltaje bajo cuando una línea de palabra seleccionada se jala abajo y existe un diodo en la intersección de la línea de bit y la línea de palabra seleccionada. Los diodos se colocan en las celdas de bit que deben almacenar 1; entonces un diodo corresponde a 0. Si los diodos se colocan en todas las intersecciones y se proporciona un mecanismo para desconectar selectivamente cada diodo no necesario, con la quema de un fusible, resulta una *ROM programable*, o PROM. La programación de una PROM se realiza al colocarla en un dispositivo especial (*programador PROM*) y aplicar corrientes para quemar fusibles seleccionados.

Una PROM borrable (EPROM) usa un transistor en cada celda que actúa como interruptor programable. Los contenidos de una EPROM se pueden borrar (fijar a todos 1) al exponer el dispositivo a luz ultravioleta durante algunos minutos. Puesto que las ROM y las PROM son más simples y más baratas que las EPROM, éstas se usan durante el desarrollo y depuración de sistemas, y los contenidos se colocan en ROM cuando los datos o programas se finalizan. El borrado de datos en PROM eléctricamente borrables (EEPROM) es tanto más conveniente como selectivo. Cualquier bit dado en la matriz de memoria se puede borrar al aplicar un voltaje apropiado a través del transistor correspondiente. Con frecuencia hay un límite (usualmente, muchos cientos) acerca de cuántas veces se pueden borrar una celda EEPROM antes de que pierda la capacidad de retener información. Sin embargo, a diferencia de la DRAM, esta carga es atrapada de modo que no puede escapar; por tanto, no requiere regeneración ni energía para mantenerlos en su lugar durante muchos años. Puesto que el borrado ocurre en bloques y es relativamente lento, las memorias tipo *flash* no sustituyen a las SRAM o DRAM; más bien, se usan principalmente para almacenar información acerca de la configuración o establecimiento por defecto de sistemas digitales que cambian de modo más bien poco frecuente y se deben preservar cuando el sistema no tiene energía.

Otras tecnologías para memorias de acceso aleatorio no volátil que parecen prometedoras incluyen la RAM ferroeléctrica, la RAM magnetorresistiva y la memoria unificada óptica [Gepp03].

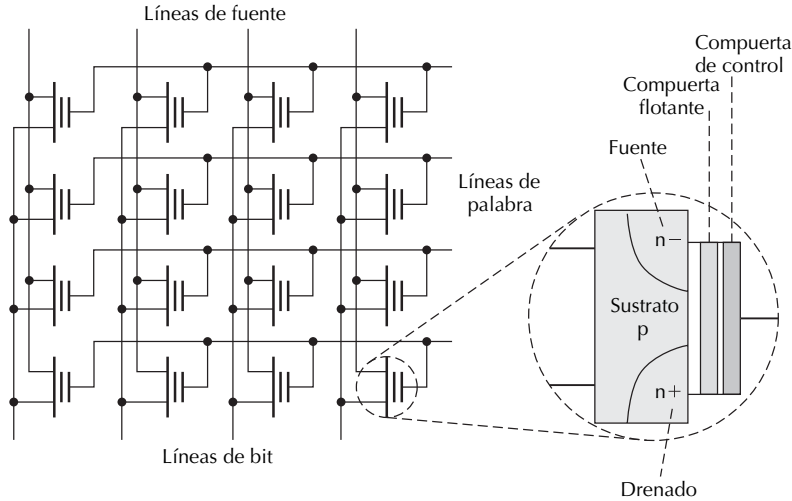


Figura 17.13 Organización EEPROM o memoria tipo *flash*. Cada celda de memoria se construye de un transistor MOS de compuerta flotante.

17.6 Necesidad de una jerarquía de memoria

Para equiparar la rapidez del procesador, la memoria que retiene las instrucciones y datos del programa debe ser accesible en 1 ns o menos. Se puede construir tal memoria, pero sólo en tamaños pequeños (por ejemplo, un archivo de registro). El volumen del programa y sus datos se deben mantener en memoria más lenta y más grande. El reto es diseñar el sistema de memoria global de modo que parezca tener la rapidez del componente más rápido y el costo más barato. Aunque aumentar el ancho de banda de la memoria lenta puede ayudar a solventar la brecha de rapidez, este enfoque requiere métodos cada vez más sofisticados para esconder la latencia de memoria y eventualmente caer cuando la brecha crezca lo suficientemente ancha. El ejemplo 17.3 muestra que el incremento del ancho de banda puede generar problemas incluso en ausencia de límites en los métodos para ocultar latencia.

Ejemplo 17.3: Relación entre ancho de banda de memoria y rendimiento Estime el ancho de banda de memoria principal mínimo requerido para sostener una tasa de ejecución de instrucciones de 10 GIPS. Suponga que no hay memoria caché u otro *buffer* rápido para instrucciones o datos. ¿Este ancho de banda sería factible con una latencia de memoria de 100 ns y una frecuencia de bus de 200 MHz?

Solución: La suposición de no caché significa que cada instrucción ejecutada se debe leer (*fetch*) de la memoria principal. De este modo, la tasa de ejecución de 10 GIPS implica al menos diez mil millones de acceso de instrucción por segundo. Si supone instrucciones de cuatro bytes, esto implica un ancho de banda de memoria mínimo de 40 GB/s, incluso si alguna instrucción no se lee (*fetch*) o almacena alguna vez. Extraer de memoria 40 B/ns a una latencia de acceso de 100 ns requeriría que en cada acceso se leyeran 4 000 B. Este ancho de acceso, aunque no es completamente factible, más bien resulta impráctico. Observe que el análisis se basa en tres suposiciones muy optimistas: carencia total de referencias de datos, habilidad para dispersar los accesos a memoria uniformemente sobre el tiempo y ocultamiento perfecto de latencia. La transferencia de 40 GB/s de la memoria al procesador sobre un bus de 200 MHz requeriría un ancho de bus de 200 B. De nuevo, la suposición de que ningún ciclo de bus se desperdicia es optimista.

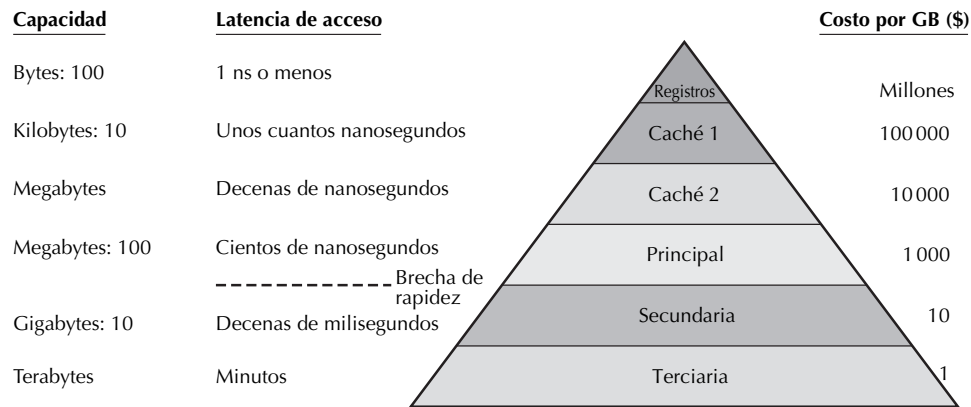


Figura 17.14 Nombres y características clave de niveles en una jerarquía de memoria.

El hecho de que se pueden construir memorias pequeñas con tiempos de acceso de no más de unos cuantos nanosegundos lleva a la idea de una *memoria buffer* que retiene los segmentos de programa y datos actualmente más útiles (aquellos a los que se hace referencia frecuente o se sabe que serán necesarios muy pronto), en forma muy parecida a como los registros retienen algunos de los elementos de datos actualmente activos para rápido acceso por el procesador. Este *buffer* o *memoria caché* quizá es demasiado pequeña para retener todos los programas y datos requeridos, de modo que los objetos se deben llevar ahí desde la memoria principal. Tal caché no salva por completo la brecha de rapidez entre el procesador y la memoria principal, de modo que con frecuencia se usa una memoria caché de segundo nivel. Estos dos niveles de caché, junto con los registros, la memoria principal, la memoria secundaria (usualmente disco) y la memoria terciaria o de archivo, se muestran en la figura 17.14.

Se dice que las memorias en interacción de diversas capacidades, rapidez y costos que se muestran en la figura 17.14 constituyen una *jerarquía de memoria* dentro de la cual los datos y componentes de programa más útiles de algún modo se mueven hacia niveles superiores, donde son accesibles más fácilmente. En la práctica, cada sistema de cómputo moderno tiene una *memoria jerárquica*, aunque no todos los niveles que se muestran en la figura 17.14 están presentes en todos los casos.

La forma piramidal de la figura 17.14 tiene la intención de presentar el tamaño más pequeño de unidades de memoria en lo alto y las mayores capacidades en el fondo. Los archivos de registro rápidos tienen capacidades que se miden en cientos o cuando mucho miles de bytes. En la actualidad, son prácticas las memorias terciarias multiterabytes, y la introducción de unidades petabyte está bajo consideración. Desde luego, el costo y la rapidez varían en la dirección opuesta, siendo las memorias más pequeñas cerca de lo alto las más rápidas y más costosas. Al examinar las latencias de acceso dadas en la figura 17.4, se observa que las razones de latencias para niveles sucesivos son bastante pequeñas cerca de la cima (diez o menos). Súbitamente, la razón crece en casi 10^5 entre las memorias principal y secundaria, y también es grande, aunque no tan mal, entre memorias secundaria y terciaria. Esta *brecha de rapidez* entre memorias semiconductoras y magneticoópticas es un importante impedimento para alto rendimiento en aplicaciones de datos intensos.

Los niveles en la jerarquía de memoria de la figura 17.14 están numerados como sigue: 0 para registros, 1 y 2 para caché (conocidos como caché L1 y L2), 3 para memoria principal, etcétera. A veces los registros se excluyen de la jerarquía de memoria, en parte porque los mecanismos para mover datos adentro y afuera de los registros (instrucciones *carga/almacenamiento* explícitas) son diferentes de los usados para transferencias de datos entre otros niveles. Además, en algunas arquitecturas, ciertos elementos de datos nunca se mueven en los registros y se procesan directamente de la memoria caché. No

obstante, en virtud de que el enfoque de este libro está sobre las arquitecturas carga/almacenamiento en las que los datos se deben cargar en registros antes de poder manipularlos, es apropiado incluir los registros como nivel 0 de la jerarquía de memoria. Observe que el archivo de registro se usa como *buffer* de gran rapidez sólo para datos. Sin embargo, muchos procesadores modernos usan *buffer* de instrucción que tiene el mismo papel en relación con las instrucciones como el que desempeña el archivo de registro con los datos.

Al nivel registro, o nivel caché 1, a veces se le considera el *nivel más alto* de la jerarquía de memoria; aunque esta nomenclatura es consistente con la vista piramidal que se muestra en la figura 17.14, puede causar cierta confusión cuando, por ejemplo, el nivel 1 se visualiza como mayor que el nivel 3. Por tanto, en este libro se evitará caracterizar los niveles de la jerarquía de memoria como “altos” o “bajos”, en vez de ello se usarán los nombres o números del nivel. Cuando se deban especificar posiciones relativas, una se caracterizará como más rápida o más cercana al procesador (respectivamente, más lenta o más alejada del procesador). Por ejemplo, cuando un objeto requerido no se encuentra en un nivel específico de la jerarquía de memoria, se consulta el siguiente nivel más lento. Este proceso de mover el foco de atención al siguiente nivel más lento continúa hasta que se encuentra el objeto. Por tanto, un objeto ubicado se copia mediante niveles sucesivamente más rápidos en su viaje hacia el procesador hasta que alcanza el nivel 0, donde es accesible al procesador.

En este capítulo se trató con la tecnología y organización básica de las unidades de memoria que comprenden los componentes principales y caché de la figura 17.14. Los registros y archivos de registro se estudiaron en el capítulo 2. Lo que resta por discutir es cómo la información se mueve de manera transparente entre las memorias caché y principal (capítulo 18), los esquemas de tecnología y almacenamiento de datos de memorias masivas empleados como componentes secundario y terciario en la jerarquía de memoria (capítulo 19) y la administración de las transferencias de información entre memorias masivas y principal (capítulo 20).

PROBLEMAS

17.1 Organización de memoria SRAM

Se proporcionan dos chips SRAM, y cada uno forma memoria de w palabras y b bits de ancho, donde b y w son pares. Muestre cómo usar una cantidad mínima de lógica externa para formar una memoria con las siguientes propiedades, o argumente que es imposible hacerlo.

- w palabras que tienen $2b$ bits de ancho
- $2w$ palabras que tienen b bits de ancho
- $w/2$ palabras que tienen $4b$ bits de ancho
- $4w$ palabras que tienen $b/2$ bits de ancho
- Almacenar copias duplicadas de w palabras que tienen b bits de ancho y comparar las dos copias durante cada acceso para detección de error

17.2 Organización de memoria SRAM

Considere chips SRAM de 4 Mb con tres organizaciones internas diferentes, que ofrecen anchos de datos de uno, cuatro y ocho bits. ¿Cuántos de cada tipo de chip

se necesitarían para construir una unidad de memoria de 16 MB con los siguientes anchos de palabra y cómo se deben interconectar?

- Palabras de ocho bits
- Palabras de 16 bits
- Palabras de 32 bits

17.3 Organización de memoria DRAM

Una encuesta de computadoras de escritorio y laptop en una organización reveló que tienen memoria principal DRAM desde 256 MB hasta 1 GB, en incrementos de 128 MB. Se ha dicho que estas máquinas usan chips DRAM con capacidades de 256 Mb o 1 Gb, y que ambos tipos nunca se mezclan en la misma máquina. ¿Qué revelan estas partes de información acerca del número de chips de memoria en las PC? En cada caso, especifique los posibles anchos de datos para los chips usados, si el ancho de palabra de memoria DRAM es de 64 bits.

17.4 Tendencias en tecnología DRAM

Suponga que las tendencias en la figura 17.7 continúan para el futuro previsible. ¿Cuál sería el rango esperado de valores para el número de chips de memoria usada y la capacidad de memoria global para:

- Estaciones de trabajo en el año 2008?
- Servidores en el 2015?
- Supercomputadoras en el 2020?
- Grandes PC en el 2010?

17.5 Diseño de una DRAM de palabra ancha

Una unidad SDRAM proporciona palabras de datos de 64 bits, toma cuatro ciclos de reloj para la primera palabra y un ciclo para cada palabra subsecuente (hasta siete más) dentro de la misma hilera de memoria. ¿Cómo podría usar esta unidad como componente para construir una SDRAM de palabra ancha con un tiempo de acceso fijo, y cuál sería la latencia de memoria resultante para anchos de palabra de 256, 512 y 1 024 bits?

17.6 Operación DRAM encauzada

En la descripción de DRAM se observó que un chip DRAM necesita la mitad de pines de dirección que una SRAM del mismo tamaño porque las direcciones de hilera y columna no tienen que proporcionarse de manera simultánea. ¿Por qué la operación encauzada de la figura 17.10 cambia esta situación?

17.7 Arquitecturas procesador en memoria

Se anticipó el argumento de que, como consecuencia de que el ancho de banda de la memoria externa se ha vuelto un serio problema, quizá se puede explotar el mayor ancho de banda interno (debido a lectura de toda una hilera de la matriz de memoria a la vez) con la implementación de muchos procesadores simples en un chip de memoria DRAM. Los procesadores múltiples pueden manipular segmentos de la larga palabra de memoria interna a la que se puede acceder en cada ciclo de memoria. Discuta brevemente los puntos positivos, y desventajas o problemas de implementación, de este enfoque para derrumbar la pared de memoria.

17.8 Memoria encauzada con interpolado

Proporcione el diseño detallado para el mecanismo de detección de conflicto y atasco necesarios para conver-

tir una unidad de memoria interpolada de cuatro vías en una memoria encauzada de cuatro etapas, como se discute en la sección 17.4.

17.9 Organización de datos en memorias interpoladas

Suponga que los elementos de dos matrices, A y B , de 1000×1000 , se almacenan en la unidad de memoria interpolada de cuatro vías de la figura 17.11 en orden a partir de la hilera mayor. Por tanto, los mil elementos de una hilera de A o B se dispersan igualmente entre los bancos, mientras que los elementos de una columna caen en el mismo banco.

- Si supone que el tiempo de cálculo es despreciable en comparación con el tiempo de acceso a memoria, derive el rendimiento total de memoria cuando los elementos de A se suman a los elementos correspondientes de B en orden a partir de la hilera mayor.
- Repita la parte a), pero esta vez suponga que las sumas se realizan en orden de mayor columna.
- Demuestre que, para algunas matrices $m \times m$, los cálculos de las partes a) y b) conducen a rendimientos totales de memoria comparables.
- Repita la parte a), pero esta vez suponga que B se almacena en orden de mayor columna.

17.10 Paso de acceso en memorias interpoladas

Una forma conveniente de analizar los efectos de la organización de datos en memorias interpoladas sobre el rendimiento total de memoria es a través de la noción de *paso de acceso*. Considere una matriz $m \times m$ que se almacena en una memoria interpolada de h vías en orden de mayor hilera. Cuando se accede a los elementos en una hilera particular de tal matriz, las localidades de memoria $x, x + 1, x + 2, \dots$ son direccionadas; se dice que el paso de acceso es 1. Los elementos de una columna caen en las localidades $y, y + m, y + 2m, \dots$, para un paso de acceso de m . Acceder a los elementos en la diagonal principal conduce a un paso de acceso de $m + 1$, mientras que los elementos de la antidiagonal producen un paso de $m - 1$.

- Demuestre que el rendimiento total de la memoria está maximizado en tanto el paso de acceso s sea relativamente primo con respecto a h .
- Una forma de garantizar el máximo rendimiento total de memoria para todos los pasos es elegir h como

número primo. ¿Por qué esto no es particularmente una buena idea?

- c) En lugar de almacenar cada hilera de la matriz que comienza con su elemento 0, se puede comenzar el almacenamiento en la hilera i con su i -ésimo elemento, regresando al comienzo de la hilera después de que aparece el último elemento de la hilera. Demuestre que este tipo de *almacenamiento sesgado* es útil para mejorar el rendimiento total de memoria en algunos casos.

17.11 Memoria de sólo lectura

Una aplicación de las memorias de sólo lectura está en la evaluación de funciones. Suponga que se quiere evaluar una función $f(x)$, donde x representa una fracción de ocho bits y el resultado se obtendrá con 16 bits de precisión.

- a) ¿Qué tamaño de ROM se necesita y cómo debe estar organizada?
- b) Dado que las memorias son más lentas que los circuitos lógicos, ¿este método alguna vez es más rápido que la evaluación de función convencional que se usa en los circuitos aritméticos de tipo ALU?

17.12 Elección de tecnología de memoria

- a) Las especificaciones de una cámara digital indican que su subsistema de almacenamiento de fotografías contiene una unidad de memoria SRAM muy grande como para retener un par de fotografías y un módulo de memoria tipo *flash* que puede almacenar unas 100 fotografías. Discuta las razones que son motivo de las elecciones de tecnologías y capacidades para los dos componentes de memoria.
- b) Repita la parte a) para un organizador electrónico, que almacene varios miles de nombres e información de contacto asociada, con la misma combinación de memorias: una unidad de memoria SRAM pequeña y un módulo de memoria *flash* muy grande.

17.13 Organización de memoria tipo *flash*

La memoria tipo *flash* es de acceso aleatorio en cuanto se refiere a la lectura. Sin embargo, para la escritura un elemento de datos arbitrario no se puede modificar en lugar porque no es posible el borrado de palabra sencilla. Estudie organizaciones de datos alternas en me-

moria tipo *flash* que permita operaciones de escritura selectiva.

17.14 Características de jerarquía de memoria

Aumente la figura 17.14 con una columna que especifique las tasas aproximadas de datos pico en los diversos niveles de la jerarquía. Por ejemplo, un archivo de registro de 32 bits y tres puertos (dos lecturas y una escritura por ciclo) con tiempo de acceso aproximado de 1 ns puede soportar una tasa de datos de $3 \times 32 \times 10^9 \text{ b/s} = 12 \text{ GB/s} \cong 0.1 \text{ Tb/s}$. Tal vez se puede aumentar esta tasa por un factor de 10, a 1 Tb/s, si los registros fuesen el doble de rápidos y el doble de anchos, y se pudiera acceder a ellos a través de unos cuantos más puertos lectura/escritura.

17.15 Características de jerarquía de memoria

Con base en la figura 17.14, ¿cuál nivel de una memoria jerárquica es probable que tenga mayor costo? Observe que se pregunta el costo total, no el costo por byte.

17.16 Analogía para la memoria jerárquica de una computadora

Considere la forma jerárquica en la que una persona trata con los números telefónicos. Tiene memorizados algunos de números telefónicos importantes. Los números para otros contactos clave se mantienen en un directorio telefónico de bolsillo o un organizador electrónico. Si en este caso se avanza más por el equivalente de la figura 17.14, se llega al directorio telefónico de la ciudad y, finalmente, a la colección de directorios de todo el país disponibles en la biblioteca local. Dibuje y etiquete adecuadamente un diagrama piramidal para representar este sistema jerárquico, y mencione las características importantes de cada nivel.

17.17 Historia de la tecnología de memoria principal

En la actualidad se dan por hecho las memorias de acceso aleatorio que son rápidas y de gran capacidad. Las modernas computadoras de escritorio y laptop tienen capacidades RAM que superan las memorias secundarias de las primeras computadoras digitales más poderosas. Éstas no tenían memorias de acceso aleatorio, más bien usaban memorias seriales en las que los objetos de

datos se movían en una ruta circular; el acceso a un objeto particular requería largas esperas hasta que el objeto aparecía en la posición correcta para su lectura. Incluso cuando la capacidad del acceso aleatorio apareció en las memorias de *núcleo magnético*, el proceso de fabricación era complicado y costoso, ello propició el uso de memorias principales muy pequeñas para los estándares actuales. Otros ejemplos de las ahora abandonadas tec-

nologías de memoria incluyen las memorias de *rodillo alambrado*, las memorias de *línea de retardo sónico* y las memorias de *burbuja magnética*. Escoja una de éstas, o alguna otra tecnología de memoria previa a la década de 1970, y escriba un informe que describa la tecnología, las organizaciones de memoria asociadas y el rango de aplicaciones prácticas en computadoras digitales y otras partes.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Crag96] Cragon, H. G., *Memory Systems and Pipelined Processors*, Jones and Bartlett, 1996.
- [Cull99] Culler, D. E. y J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1999, Sec. 12.1, pp. 936-955.
- [Cupp01] Cuppu, V., B. Jacob, B. Davis y T. Mudge, "High-Performance DRAMS in Workstation Environments", *IEEE Trans. Computers*, vol. 50, núm. 11, pp. 1133-1153, noviembre de 2001.
- [Gepp03] Geppert, L., "The New Indelible Memories", *IEEE Spectrum*, vol. 40, núm. 3, pp. 49-54, marzo de 2003.
- [Kush91] Kushiya, N., Y. Watanabe, T. Ohsawa, K. Muraoka, Y. Nagahama, and T. Furuyama, "A 12-MHz Data Cycle 4-Mb DRAM with Pipeline Operation", *IEEE J. Solid-State Circuits*, vol. 26, No. 4, pp. 479-482, 1991.
- [Przy94] Przybylski, S., *New DRAM Technologies*, MicroDesign Resources, 1994.
- [Shri98] Shriver, B. y B. Smith, *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*, IEEE Computer Society Press, 1998, Cap. 5, pp. 427-461.
- [Wake01] Wakerly, J. F., *Digital Design: Principles and Practices*, 3a. ed., Prentice Hall, 2001.
- [Wulf95] Wulf, W. A. y S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious", *Computer Architecture News*, pp. 20-24, marzo de 1995.

ORGANIZACIÓN DE MEMORIA CACHÉ

“Se discute el uso de una memoria central rápida... esclavizada a otra más lenta... de manera tal que en casos prácticos el tiempo de acceso efectivo está más cerca de la memoria rápida que de la lenta.”

Maurice Wilkes, Memorias esclavas y asignación de almacenamiento dinámico, 1965.

“La memoria representa el gabinete de la imaginación, el tesoro de la razón, el registro de la conciencia y la cámara de consejo del pensamiento.”

Saint Basil

TEMAS DEL CAPÍTULO

- 18.1** La necesidad de una caché
- 18.2** ¿Qué hace funcionar a una caché?
- 18.3** Caché de mapeo directo
- 18.4** Caché de conjunto asociativo
- 18.5** Memoria caché y memoria principal
- 18.6** Mejoramiento del rendimiento caché

La cita de Maurice Wilkes muestra que la idea de usar una memoria rápida (esclava o caché) para salvar la brecha de rapidez entre un procesador y una memoria principal más lenta, pero mayor, se ha discutido durante algún tiempo. Aun cuando en los años intermedios, las memorias se han vuelto más rápidas, la brecha de rapidez procesador-memoria se ha ensanchado tanto que el uso de una memoria caché ahora casi es obligatorio. La relación de caché a memoria principal es similar a la de un cajón de escritorio a un archivero: un lugar más fácilmente accesible para retener datos de interés actual durante la duración de tiempo cuando sea probable que se ingrese a los datos. En este capítulo se revisan estrategias para mover datos entre memorias principal y caché, así como aprender formas de cuantificar la resultante mejora en rendimiento.

■ 18.1 La necesidad de una caché

La latencia de acceso a memoria constituye un gran obstáculo de rendimiento en las computadoras modernas. Las mejoras en tiempo de acceso a memoria (actualmente miles de nanosegundos para memorias fuera de chip grandes) no mantuvo el paso de la rapidez de los procesadores (< 1 ns a unos cuantos nanosegundos por operación).

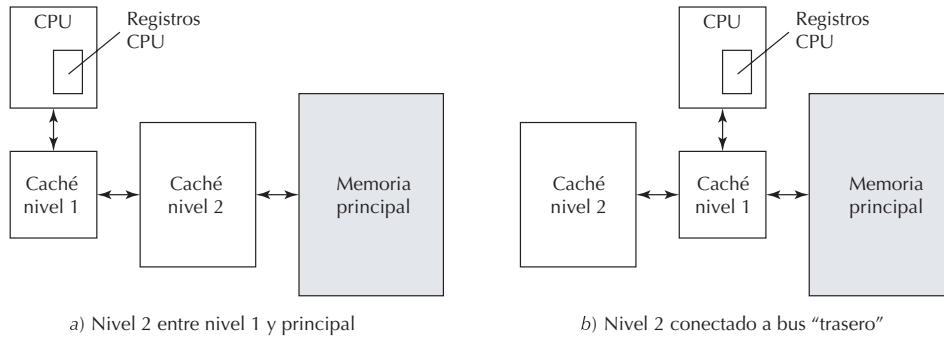


Figura 18.1 Las memorias caché actúan como intermediarias entre el procesador superrápido y la memoria principal mucho más lenta.

Como consecuencia de que las memorias más grandes tienden a ser más lentas, el problema empeora por los tamaños de memoria cada vez mayores. En este contexto, la mayoría de los procesadores usan una memoria rápida relativamente pequeña para mantener instrucciones y datos de modo que se evite la mayoría de los accesos de tiempo a la memoria principal lenta. Los datos todavía deben moverse de la memoria más lenta/grande a la memoria más pequeña/rápida y viceversa, pero se puede traslapar con el programa de cálculo. El término “caché” (lugar de ocultamiento seguro) se usa para esta memoria pequeña/rápida porque usualmente es invisible al usuario, excepto a través de su efecto sobre el rendimiento.

En la actualidad es muy común el uso de una memoria caché de segundo nivel para reducir los accesos a la memoria principal. La figura 18.1 muestra la relación entre memorias caché y principal. La memoria caché de nivel 2 se puede añadir a un sistema con memorias caché y principal mediante dos formas. La primera opción (figura 18.1a) consiste en insertar la memoria caché de nivel 2 entre la memoria caché de nivel 1 y la memoria principal. La segunda opción conecta la unidad de caché de nivel 1 tanto a la caché de nivel 2 como a la memoria principal directamente a través de dos buses de memoria diferentes (a veces llamados buses trasero y frontal).

Considere una memoria caché de un solo nivel. Para ingresar una palabra de datos requerida, primero se consulta la caché. Al hecho de encontrar estos datos se le denomina *impacto de caché*; no encontrarlos significa un *fallo de caché*. Un parámetro importante en la evaluación de la efectividad de las memorias caché es la *tasa de impacto (hit rate)*, que se define como la fracción de accesos de datos que se pueden satisfacer desde la caché en contraste con la memoria más lenta que se asienta más allá de ella. Por ejemplo, una tasa de impacto de 95% significa que sólo uno en 20 accesos, en promedio, no encontrará los datos requeridos. Con una tasa de impacto h , ciclo de acceso a caché de $C_{\text{rápido}}$ y ciclo de acceso a memoria más lento C_{lento} , el tiempo de ciclo de memoria efectivo es

$$C_{\text{efectivo}} = hC_{\text{rápido}} + (1 - h)(C_{\text{lento}} + C_{\text{rápido}}) = C_{\text{rápido}} + (1 - h)C_{\text{lento}}$$

Esta ecuación se deriva con la suposición de que, cuando los datos no se encuentran en la caché, primero se deben llevar a esta última (en el tiempo C_{lento}) y luego se accede a ellos desde la caché (en el tiempo $C_{\text{rápido}}$). El adelantamiento simultáneo de datos desde la memoria lenta hacia tanto el procesador como a la caché, reduce un poco el retardo efectivo, pero la fórmula simple para C_{efectivo} es adecuada para estos propósitos, en especial como consecuencia de que no se toma en cuenta la cabecera de determinar si los datos requeridos están en la caché. Se ve que, cuando la tasa de impacto h está cerca de 1, se logra un ciclo de memoria efectivo cercano a $C_{\text{rápido}}$. Por tanto, la caché ofrece la ilusión de que todo el espacio de memoria consta de memoria rápida.

En un microprocesador típico, el acceso a memoria caché forma parte del ciclo de ejecución de instrucciones. En tanto los datos requeridos están en la caché, la ejecución de instrucción continúa a toda rapidez. Cuando ocurre un fallo de caché y se debe acceder a la memoria más lenta, la ejecución de instrucción se interrumpe. La *penalización por fallo de caché* usualmente se especifica en términos del número de ciclos de reloj que se desperdiciarán porque el procesador tiene que atascarse hasta que los datos estén disponibles. En un microprocesador que ejecuta, en promedio, una instrucción por ciclo de reloj, cuando no hay fallo de caché, una penalización por fallo de caché de ocho ciclos significa que éstos se sumarán al tiempo de ejecución de instrucción. Si 5% de las instrucciones encuentran un fallo de caché, correspondiente a una tasa de impacto caché de 95%, y si se supone un promedio de un acceso a memoria por instrucción ejecutada, el CPI efectivo será $1 + 0.05 \times 8 = 1.4$.

Cuando se tiene una caché de segundo nivel o L2, se le puede asociar una tasa impacto o fallo local. Por ejemplo, una tasa de impacto local de 75% para la caché L2 significa que 75% de los accesos que se refirieron a la caché L2 (debido a un fallo en la caché L1) se puede satisfacer aquí; en este sentido, 25% necesita una referencia a la memoria principal.

Ejemplo 18.1: Rendimiento de un sistema caché de dos niveles Un sistema de cómputo tiene cachés L1 y L2. Las tasas de impacto local para L1 y L2 son 95 y 80%, respectivamente. Las penalizaciones por fallos son ocho y 60 ciclos, de manera respectiva. Si supone un CPI de 1.2 sin algún fallo de caché y promedio de 1.1 accesos a memoria por instrucción, ¿cuál será el CPI efectivo después de factorizar los fallos de caché? Para el caso de que los dos niveles de caché se toman como una sola memoria caché, ¿cuáles serán su tasa de fallo y penalización por fallo?

Solución: Se puede usar la fórmula $C_{\text{efectivo}} = C_{\text{rápido}} + (1 - h_1)[C_{\text{medio}} + (1 - h_2)C_{\text{lento}}]$. Puesto que $C_{\text{rápido}}$ ya está incluido en el CPI de 1.2, se debe tomar en cuenta el resto. Esto último conduce a un CPI efectivo de $1.2 + 1.1(1 - 0.95)[8 + (1 - 0.8)60] = 1.2 + 1.1 \times 0.05 \times 20 = 2.3$. Cuando las dos cachés se ponen juntas, se tiene una tasa de impacto de 99% (tasa de impacto de 95% en nivel 1, más 80% de los 5% fallos del nivel 1 que se encuentran en el nivel 2) o una tasa de fallo de 1%. El tiempo de acceso efectivo de esta imaginaria caché de un solo nivel es $1 + 0.05 \times 8 = 1.4$ ciclos y su penalización por fallo es de 60 ciclos.

Una memoria caché se caracteriza por diversos parámetros de diseño que influyen su costo de implementación y rendimiento (tasa de impacto). La siguiente descripción se refiere a la suposición de un solo nivel de caché; esto es, no hay caché de nivel 2. Los más importantes parámetros de caché son:

- Tamaño de caché* en bytes o palabras. Una caché más grande puede retener más datos útiles del programa, pero es más costosa y quizá más lenta.
- Tamaño de bloque o ancho de línea de caché*, que se define como la unidad de transferencia de datos entre el caché y la memoria principal. Con una línea de caché más grande, se llevan más datos a la caché con cada fallo. Lo anterior puede mejorar la tasa de impacto, pero también tiende a ligar partes de la caché con datos de menor utilidad.
- Política de colocación*. Determinar dónde se puede almacenar una línea de caché entrante. Políticas más flexibles implican mayor costo de hardware y puede o no tener beneficios de rendimiento como consecuencia de sus procesos más complejos y lentos, para alojar los datos requeridos en la caché.

- d) *Política de sustitución*. Determinar cuál de los muchos bloques de caché (en los que se puede mapear una nueva línea de caché) se debe sobrescribir. Las políticas típicas incluyen la elección de un bloque aleatorio y del bloque menos utilizado.
- e) *Política de grabación*. Determinar si las actualizaciones a palabras de caché se adelantan inmediatamente a la memoria principal (política de *grabación directa*) o bloques de caché modificados se copian a la memoria principal en su totalidad y cuándo deben sustituirse en la caché (política de *escritura inversa* o *copy-back*).

Estos parámetros están cercanamente relacionados; el hecho de que se cambie uno con frecuencia significa que los otros también necesitan cambios para asegurar un rendimiento óptimo de memoria. El impacto de estos parámetros en el rendimiento del sistema de memoria se verá en la sección 18.6.

■ 18.2 ¿Qué hace funcionar a una caché?

Las cachés son tan exitosas en el mejoramiento del rendimiento de los procesadores modernos debido a dos propiedades de localidad de patrones de acceso a memoria en programas típicos. La *localidad espacial* de los accesos de memoria resulta de accesos consecutivos que se refieren a localidades de memoria cercanas. Por ejemplo, un ciclo de programa de nueve instrucciones que se ejecuta mil veces provoca que los accesos de instrucción se concentren en una región de nueve palabras del espacio de dirección para un periodo largo (figura 18.2). De igual modo, conforme un programa se ensambla, la tabla de símbolos, que ocupa un sitio pequeño en el espacio de dirección, se consulta con frecuencia. La *localidad temporal* indica que cuando se accesa a una instrucción o elemento de datos, los accesos futuros al mismo objeto tienden a ocurrir principalmente en el futuro cercano. En otras palabras, los programas tienden a enfocarse en una región de memoria para obtener instrucciones o datos y luego moverse hacia las otras regiones según se completan las fases de cálculo.

Las dos propiedades de localidad de patrones de acceso a memoria propician que las instrucciones y elementos de datos más útiles en algún punto específico en la ejecución de un programa (a veces

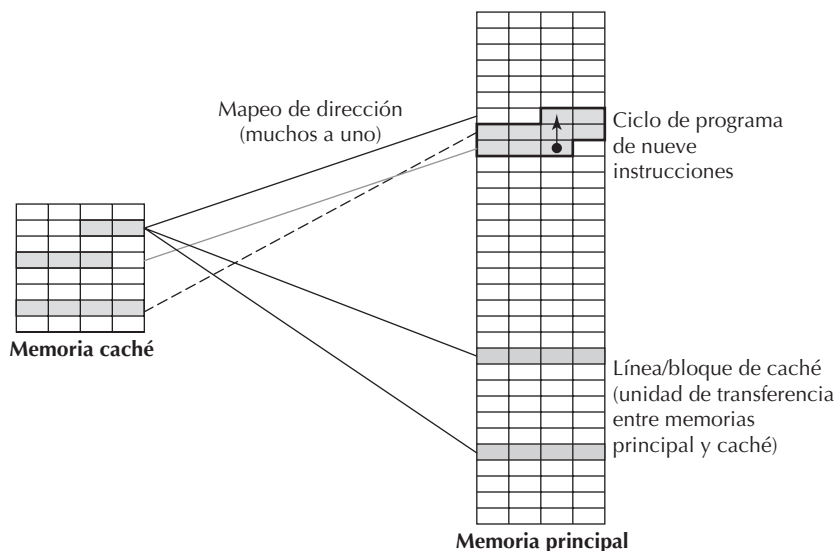


Figura 18.2 Si se supone que no hay problema alguno en el mapeo de dirección, la caché retendrá un ciclo de pequeño programa en su totalidad, ello conduce a rápida ejecución.

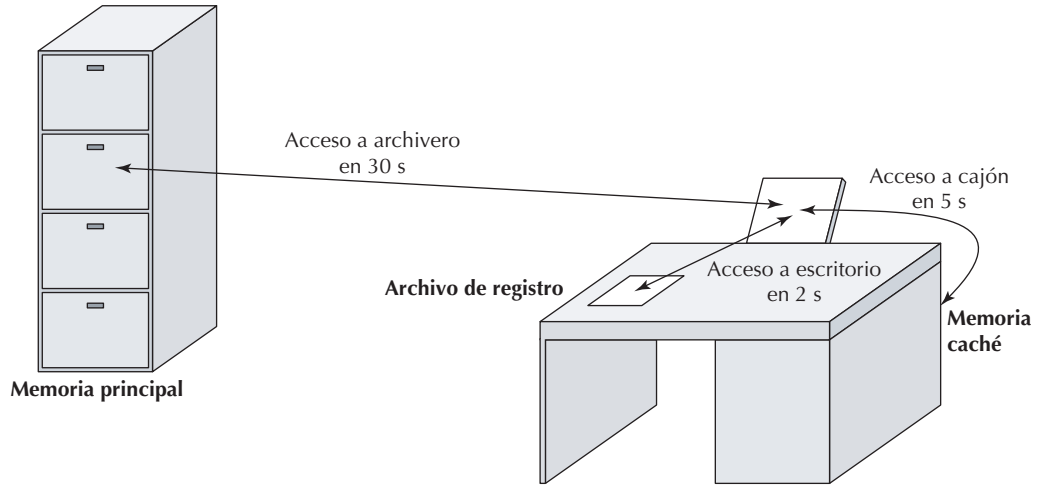


Figura 18.3 Los objetos sobre un escritorio (registro) o en un cajón del escritorio (caché) son más accesibles que los que se encuentran en un archivero (memoria principal).

llamado el *juego operativo* del programa) reside en la caché. Esto último conduce a altas tasas de impacto de caché, que usualmente están en el rango de 90 a 98%, o, de manera equivalente, a bajas tasas de fallos, en el rango de 2 a 10%.

La siguiente analogía es muy útil (figura 18.3). Usted trabaja en los documentos que coloca en su escritorio. El escritorio corresponde a un registro del CPU. Entre los documentos o archivos que no necesita ahora, los más útiles, se encuentran los que pueden estar en un cajón del escritorio (analogía con la memoria caché) y el resto en archiveros (memoria principal) o en una bodega (memoria secundaria). La mayor parte del tiempo encuentra lo que necesita en su cajón. De cuando en cuando tiene que ir a leer (*fetch*) al archivero los documentos o archivos que usa con menos frecuencia; en raras ocasiones es posible que tenga que ir a la bodega para leer (*fetch*) un documento o archivo pocas veces consultado. Con las latencias dadas en la figura 18.3, si la tasa de impacto en el cajón es de 90%, el tiempo promedio de acceso al documento es $5 + (1 - 0.90)30 = 8$ s.

Ejemplo 18.2: Analogía entre cajón y archivero Si supone una tasa de impacto de h en el cajón, formule la situación que se muestra en la figura 18.2 en términos de la ley de Amdahl y la aceleración que resulta del uso de un cajón.

Solución: Sin el cajón, un documento se accede a 30 s. Por tanto, leer (*fetch*) mil documentos tomaría, por decir, 30 mil s. El cajón hace que una fracción h de los casos se realice seis veces más rápido, y el tiempo de acceso para los restantes $1 - h$ permanece invariable. Por tanto, la aceleración es $1/(1 - h + h/6) = 6/(6 - 5h)$. Mejorar el tiempo de acceso al cajón puede aumentar el factor de aceleración, pero en tanto la tasa de fallos permanezca en $1 - h$, la aceleración nunca puede superar $1/(1 - h)$. Dado que $h = 0.9$, por ejemplo, la aceleración que se logra es 4, siendo el límite superior 10 para un tiempo de acceso a cajón muy corto. Si todo el cajón se pudiera colocar en el escritorio, entonces se lograría un factor de aceleración de $30/2 = 15$ en 90% de los casos. La aceleración global sería $1/(0.1 + 0.9/15) = 6.25$. Sin embargo, observe que éste es un análisis hipotético; ¡apilar documentos y archivos en su escritorio no es una buena forma de mejorar la rapidez de acceso!

Es conveniente revisar brevemente los tipos de fallo de caché y ubicarlos como fallos obligatorios, de capacidad y de conflicto.

Fallos obligatorios: El primer acceso a cualquier línea de caché resulta en un fallo. Algunos fallos “obligatorios” pueden evitarse al predecir acceso futuro a objetos y leerlos previamente (*prefetching*) en el caché. De este modo, tales fallos realmente son obligatorios sólo si se usa una *política de lectura (fetching) a petición*. A estos fallos a veces se les conoce como *fallos de arranque en frío*.

Fallos de capacidad: Como consecuencia de que el tamaño de la memoria caché es finito, se tienen que desechar algunas líneas de ésta para dejar espacio a otras; esto último conduce a fallos en lo futuro, en los que no se habría incurrido con una caché infinitamente grande.

Fallos de conflicto: Ocasionalmente existe espacio libre u ocupado por datos inútiles, en la caché, pero el esquema de mapeo usado para colocar elementos en la caché fuerza a desplazar datos útiles para colocar otros datos requeridos. Lo anterior puede conducir a fallos en lo futuro. A éstos se les denomina *fallos de colisión*.

Los fallos obligatorios son muy fáciles de entender. Si un programa tiene acceso a tres líneas diferentes de caché, en consecuencia encontrará tres fallos obligatorios. Para ver la diferencia entre los fallos de capacidad y de conflicto, considere una caché de dos líneas y los patrones de acceso $A B C A C A B$, donde cada letra representa una línea de datos. Primero, A y B se cargan en la caché. Luego se carga C (el tercero y último fallo obligatorio), que debe sustituir a A o B . Si el mapeo de caché es tal que C sustituye a A , y viceversa, entonces los siguientes tres fallos son fallos de conflicto. En este caso hay seis fallos en total (tres obligatorios, tres de conflicto). Por otra parte, si C sustituye a B , habrá cuatro fallos en total (tres obligatorios, uno de capacidad). Se podría argumentar que uno de los tres fallos adicionales en el primer ejemplo se debe ver como fallo de capacidad, pues posiblemente una caché de dos líneas no puede retener tres diferentes líneas de datos a través de sus segundos accesos. En consecuencia, ¡no tome esta categorización demasiado en serio!

Ejemplo 18.3: Fallos obligatorio, de capacidad y de conflicto Un programa tiene acceso 10 veces a cada elemento de una matriz $1\,000 \times 1\,000$ durante su ejecución. Sólo 1% de esta matriz encaja en la caché de datos en algún momento específico. Una línea de caché retiene cuatro elementos de matriz. ¿Cuántos fallos obligatorios de caché de datos causará la ejecución de este programa? Si a los 10^6 elementos de la matriz se accede una vez en la ronda i antes de comenzar la ronda de acceso $i + 1$, ¿cuántos fallos de capacidad habrá? ¿Es posible no tener fallos de conflicto en absoluto, sin importar el esquema de mapeo usado?

Solución: Llevar todos los elementos de la matriz a la caché requiere la carga de $10^6/4$ líneas de caché; por tanto, habrá 250 000 fallos obligatorios. Desde luego, es posible tener un fallo de caché hasta el primer acceso a cada uno de los 10^6 elementos de matriz. Esto último ocurriría si, cuando una línea es invocada, sólo se accedería a uno de sus cuatro elementos antes de que la línea se sustituya. Sin embargo, no todos estos fallos son obligatorios. Como para los fallos de capacidad, cada una de las diez rondas de accesos genera al menos 250 000 fallos. En consecuencia, habrá $250,000 \times 9 = 2.25 \times 10^6$ fallos de capacidad. Aunque más bien improbable, es posible que todos los accesos a cada elemento de matriz ocurran mientras residen en caché después de que se invocan por primera vez. En este caso no hay fallo de conflicto (o incluso de capacidad).

Dada una caché de tamaño fijo, dictada por factores de costo o disponibilidad de espacio en el chip procesador, los fallos obligatorios y de capacidad están bastante fijos. Por otra parte, los fallos de con-

flicto están influidos por el esquema de mapeo de datos, que está bajo su control. En las siguientes dos secciones se discutirán dos esquemas de mapeo populares.

18.3 Caché de mapeo directo

Por simplicidad, se supone que la memoria es direccionable por palabra. Para una memoria direccionable por byte, del tipo que se usa en MiniMIPS y MicroMIPS, “palabra(s)” se debe sustituir por “byte(s)”.

En el esquema de mapeo más simple, cada línea de la memoria principal tiene un lugar único en la caché donde puede residir. Suponga que la caché contiene 2^L líneas, cada una con 2^W palabras. Entonces, los bits W menos significativos en cada dirección especifican el índice de palabra dentro de una línea. Tomar los siguientes L bits en la dirección como el número de línea en la caché, causará que líneas sucesivas se mapeen a líneas sucesivas de caché. Todas las palabras cuyas direcciones tienen el mismo módulo 2^{L+W} se mapearán a la misma palabra en caché. En el ejemplo de la figura 18.4, se tiene $L = 3$ y $W = 2$; de modo que los cinco bits menos significativos de la dirección identifican una línea de caché y una palabra en dicha línea ($3 + 2$ bits) desde los cuales se debe leer la palabra deseada. En virtud de que muchas líneas de memoria principal se mapean en la misma línea de caché, ésta almacena la parte de etiqueta de la dirección para indicar cuál de las muchas líneas posibles en realidad está presente en la caché. Desde luego, una línea de caché particular puede contener datos no útiles; eso se indica mediante el restablecimiento del “bit válido” asociado con la línea de caché.

Cuando una palabra se lee desde la caché, su etiqueta también se lee (*fetch*) y compara con la etiqueta de la palabra deseada. Si coinciden, y se establece el bit válido de la línea de caché (esto es equivalente a hacer coincidir $\langle 1, \text{Tag} \rangle$ con $\langle x, y \rangle$ leído desde el almacenamiento de bit/tag válido), se indica un impacto de caché y se usa la palabra apenas leída. De otro modo, se tiene un fallo de caché y se ignora la palabra ingresada. Observe que esta lectura y coincidencia de las tags es útil para una operación de

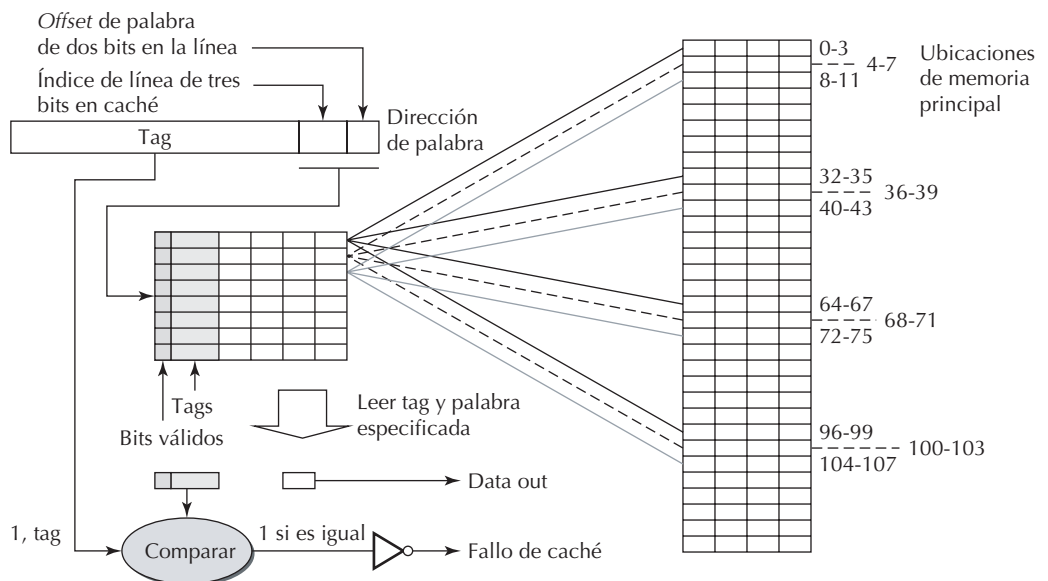


Figura 18.4 Caché de mapeo directo que contiene 32 palabras dentro de ocho líneas de cuatro palabras. Cada línea se asocia con una tag y un bit válido.

escritura. Para el caso de operación de escritura, si las tags coinciden se modifica la palabra y se le escribe de vuelta como es usual. Para el caso de equívocos, referidos como “fallo de escritura”, se debe llevar a caché una nueva línea de caché que contenga la palabra deseada (para los fallos de lectura), el proceso se realiza con la nueva línea.

El proceso de derivar la dirección caché a leer desde la dirección de palabra proporcionada se conoce como traducción de dirección. Con el mapeo directo, este proceso de traducción esencialmente es trivial y consiste en tomar los $L + W$ bits menos significativos de la dirección de palabra y usarla como la dirección de caché. Como consecuencia, en la figura 18.4 los cinco bits menos significativos de la dirección de palabra se usan como la dirección caché.

Los fallos de conflicto pueden ser un problema para cachés de mapeo directo. Por ejemplo, si a la memoria se accede con un *paso* que sea múltiplo de 2^{L+W} , cada acceso conduce a un fallo de caché. Esto ocurriría, por ejemplo, si una matriz de m columna se almacena en orden de mayor hilera y m es un múltiplo de 2^{L+W} (32 en el ejemplo de la figura 18.4). Dado que los pasos que son potencias de 2 son muy comunes, tal ocurrencia no es rara.

Ejemplo 18.4: Acceso a caché de mapeo directo Suponga que la memoria es direccionable por byte, que las direcciones de memoria tienen 32 bits de ancho, que una línea de caché contiene $2^W = 16$ bytes y que el tamaño de caché es de $2^L = 4\,096$ líneas (64 KB). Muestre las distintas partes de la dirección e identifique cuál segmento de la dirección se usa para acceder a la caché.

Solución: El *offset* de byte en una línea tiene $\log_2 16 = 4$ bits de ancho, y el índice de línea caché tiene $\log_2 4\,096 = 12$ bits de ancho. Esto deja $32 - 12 - 4 = 16$ bits para la etiqueta. La figura 18.5 muestra el resultado.

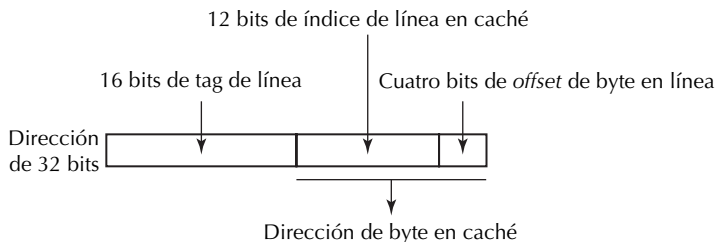


Figura 18.5 Componentes de la dirección de 32 bits en un ejemplo de caché de mapeo directo con direccionamiento por byte.

La primera cita al comienzo de este capítulo es de una breve ponencia que describe por primera vez un caché de mapeo directo, aunque el término “caché” y las implementaciones reales de la idea no aparecieron sino hasta hace pocos años. Las cachés de mapeo directo eran comunes en las primeras implementaciones; las cachés más modernas usan el esquema de mapeo de conjunto asociativo, que se describe a continuación.

■ 18.4 Caché de conjunto asociativo

Una *caché asociativa* (a veces denominada totalmente asociativa) es aquella en la que una línea de caché se puede colocar en cualquier localidad de caché; por tanto, elimina los fallos de conflicto. Tal caché es muy difícil de implementar porque necesita comparar miles de tags contra la tag deseada

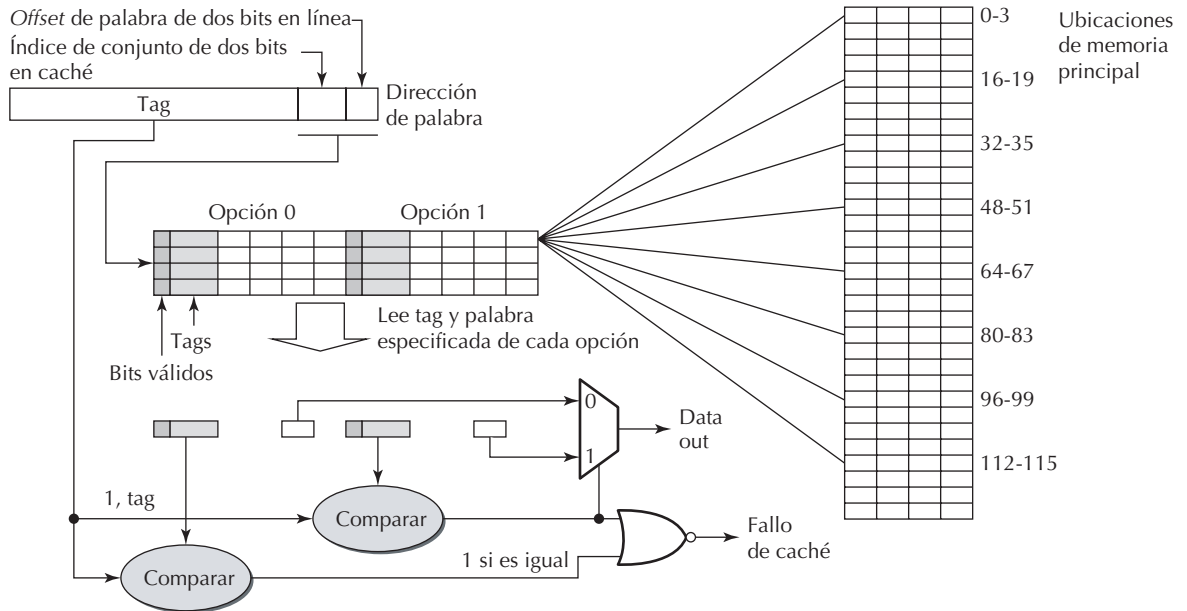


Figura 18.6 Caché de conjunto asociativo de dos vías que contiene 32 palabras de datos dentro de líneas de cuatro palabras y conjuntos de dos líneas.

para cada acceso. En la práctica, un compromiso entre caché asociativo y mapeo directo funciona muy bien. Se llama *caché de mapeo asociativo*.

El mapeo de conjunto asociativo representa el esquema de mapeo más usado en las cachés del procesador. En el extremo de los conjuntos de bloque sencillo, una caché de conjunto asociativo degenera en una caché de mapeo directo. En la figura 18.6 se muestra una operación de lectura de una caché de conjunto asociativo con un tamaño de conjunto $2^S = 2$. La dirección de memoria proporcionada por el procesador está compuesta de las partes tag e índice. Esta última, que en sí misma consta de una dirección bloque y *offset* de palabra dentro del bloque, identifica un conjunto de 2^S palabras caché que potencialmente pueden contener los datos requeridos, mientras que la tag especifica una de las muchas líneas de caché en el espacio de dirección que se mapea en el mismo conjunto de 2^S líneas caché mediante la política de colocación de conjunto asociativo. Para cada acceso a memoria se leen las 2^S palabras candidatas, junto con las tags asociadas con sus líneas respectivas. Entonces las 2^S tags se comparan simultáneamente con la tag deseada, ello conduce a dos posibles resultados:

1. Ninguna de las tags almacenadas coincide con la tag deseada: se ignoran las partes de datos y se postula una señal de fallo de caché para iniciar una transferencia de línea caché desde la memoria principal.
2. La i -ésima tag almacenada, que corresponde a la opción de colocación i ($0 \leq i < 2^S$), coincide con la tag deseada: se elige como salida de datos la palabra leída del bloque correspondiente a la i -ésima opción de colocación.

Como en la caché de mapeo directo, cada línea de caché tiene un *bit válido* que indica si retiene datos válidos. Éste también se lee junto con la tag y se usa en la comparación para asegurar que sólo ocurre una coincidencia con la tag de datos válidos. Una línea de caché de reescritura también puede tener un *bit sucio* que se fija a 1 con cada actualización de escritura a la línea y se usa al momento de sustituir la

línea para decidir si una línea que se sobrescribirá requiere respaldarse en memoria principal. Debido a las múltiples opciones de colocación para cada línea caché, los fallos de conflicto son menos problemáticos aquí que en las caché de mapeo directo.

Un punto que queda pendiente es la elección de una de las 2^S líneas caché en un conjunto a sustituir con una línea entrante. En la práctica, funcionan bien la selección aleatoria y la selección con base en cuál línea fue la menos usada (LRU, por sus siglas en inglés). Los efectos de la política de sustitución en el rendimiento de caché se discuten en la sección 18.6.

Ejemplo 18.5: Acceso a la caché de conjunto asociativo Suponga que la memoria es direccionable por byte, que las direcciones de memoria tienen 32 bits de ancho, que una línea de caché contiene $2^W = 16$ bytes, que los conjuntos contienen $2^S = 2$ líneas y que el tamaño de caché es $2^L = 4096$ líneas (64 KB). Muestre las diversas partes de la dirección e identifique cuál segmento de la dirección se usa para ingresar a la caché.

Solución: El *offset* de byte en una línea tiene $\log_2 16 = 4$ bits de ancho y el índice del conjunto en caché tiene $\log_2 (4096/2) = 11$ bits de ancho. Esto deja $32 - 11 - 4 = 17$ bits para la tag. La figura 18.7 muestra el resultado.

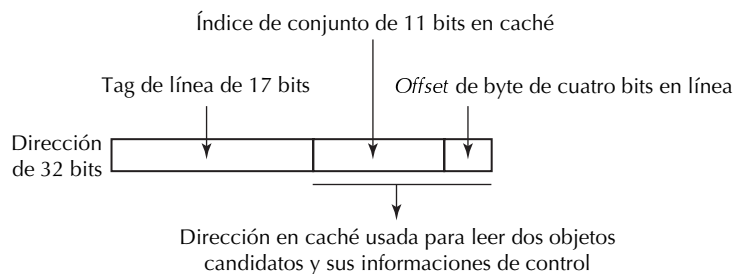


Figura 18.7 Componentes de la dirección de 32 bits en un ejemplo de caché de conjunto asociativo de dos vías.

Ejemplo 18.6: Mapeo de dirección de caché Una memoria caché de conjunto asociativo de cuatro vías y 64 KB es direccionable por byte y contiene 32 B líneas. Las direcciones de memoria tienen 32 bits de ancho.

- ¿Cuál es el ancho de las tags en esta caché?
- ¿Cuáles direcciones de memoria principal se mapean al número de conjunto 5 en la caché?

Solución: El número de conjuntos en la caché es $64 \text{ KB} / (4 \times 32 \text{ B}) = 512$.

- Una dirección de 32 bits se divide en un *offset* de byte de cinco bits, un índice de conjunto de nueve bits y una tag de 18 bits.
- Las direcciones que tienen sus índices de conjunto de nueve bits igual a cinco se mapean en el número de conjunto cinco. Dichas direcciones tienen la forma $2^{14}a + 2^5 \times 5 + b$, donde a es un valor de tag ($0 \leq a \leq 2^{18} - 1$) y b es un *offset* de byte ($0 \leq b \leq 31$). De este modo, las direcciones mapeadas al número de conjunto 5 incluyen de la 160 a la 191, de la 16 544 a la 16 575, de la 32 928 a la 32 959, etcétera.

Como es evidente a partir de la figura 18.6, las caché de conjunto asociativo de dos vías se implementan fácilmente. La política de sustitución LRU requiere un solo bit asociado con cada conjunto que

designa cuál de las dos rendijas u opciones se usaron menos recientemente. Aumentar el grado de asociatividad mejora el rendimiento de caché al reducir los fallos de conflicto, pero también complica el diseño y potencialmente alarga el ciclo de acceso. Por ende, en la práctica, el grado de asociatividad casi supera 16 y con frecuencia se mantiene en cuatro u ocho. En la sección 18.6 se verá más acerca de este tema.

18.5 Memoria caché y principal

Las cachés están organizadas de varias formas para satisfacer los requerimientos de rendimiento de memoria de máquinas específicas. Ya se ha hecho alusión a las cachés de un solo nivel y a las multiniveles (L1, L2 y, acaso, L3), y la última se bosqueja de diferentes maneras (figura 18.1). Una *caché unificada* es aquella que contiene tanto instrucciones como datos, mientras que una *caché dividida* consta de cachés de instrucciones y de datos separadas. Puesto que las primeras máquinas diseñadas en la Universidad de Harvard tenían memorias de instrucciones y datos separadas, se dice que una máquina con caché dividida sigue la *arquitectura Harvard*. Las cachés unificadas representan “encarnaciones” de la *arquitectura von Neumann*. Cada nivel de caché se puede unificar o dividir, pero la combinación más común en los procesadores de alto rendimiento consta de una caché L1 dividida y una caché L2 unificada.

A continuación, se explora la relación entre una memoria rápida (caché) y otra lenta (principal). Tópicos iguales se aplican a los métodos y requerimientos de transferencia de datos entre L1 y L2, o L2 y L3, si existen niveles de caché adicionales. Observe que usualmente es el caso de que los contenidos de un nivel dentro de la jerarquía de memoria constituye un subconjunto de los contenidos del siguiente nivel más lento. Por tanto, la búsqueda de elementos de datos procede en la dirección de memorias más lentas, lejos del procesador, porque cualquier cosa que no esté disponible en un nivel dado, no existirá en uno de los niveles más rápidos. Si un nivel de caché, por decir L1, está dividido, entonces cada una de ambas partes tiene una relación similar con el siguiente nivel caché más lento o con la memoria principal, con la complicación adicional de que las dos partes compiten por el ancho de banda de la memoria más lenta.

Puesto que la memoria principal es mucho más lenta que la memoria caché, se han desarrollado muchos métodos para hacer más rápidas las transferencias de datos entre las memorias principal y caché. Observe que las DRAM tienen anchos de banda muy altos internamente (figura 18.8). Sin embargo, este ancho de banda se pierde como resultado de las limitaciones de pin I/O en los chips de memoria y, en menor medida, a los buses relativamente estrechos que conectan la memoria principal con el caché CPU.

Considere una memoria principal de 128 MB y cuatro chips construida de los chips que se muestran en la figura 18.8. A esta memoria se puede ingresar palabras de 32 bits y transmitir las sobre un bus del mismo ancho a la caché.

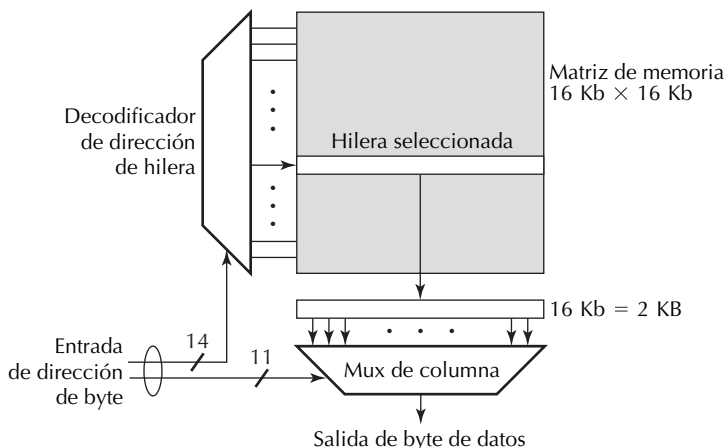


Figura 18.8 Un chip DRAM de 256 Mb organizado como un módulo de memoria 32 M x 8: cuatro de estos chips podrían formar una unidad de memoria principal de 128 MB.

Suponga que la línea de caché tiene cuatro palabras de ancho, que estarán contenidas en la misma hilera de los cuatro chips. Por tanto, cuando se haya transferido la primera palabra, las subsecuentes se leerán más rápido porque esto último se hará desde el *buffer* de hilera y no en la matriz de memoria. En virtud de que un fallo de caché ocurre cuando se ingresa una palabra particular, es posible diseñar el mecanismo de transferencia de datos para que la memoria principal lea la palabra solicitada primero, de modo que la caché suministre la palabra al CPU mientras las restantes tres palabras se transfieren. Optimizaciones como ésta son comunes en diseños de alto rendimiento.

Los procesadores de mayor rendimiento utilizan memorias principales más grandes compuestas de muchos chips DRAM. En este caso, los chips ofrecen acceso a muchas palabras (superpalabras), ello hace factible el uso de uno solo o un número pequeño de ciclos en un bus ancho para transferir toda una línea caché al CPU.

Escribir entradas de caché modificadas a la memoria principal presenta un desafío de diseño especial en los sistemas de alto rendimiento. La política de escritura (vea el final de la sección 18.1) es particularmente problemática porque, a menos que las operaciones de escritura sean poco frecuentes, hacen lenta la caché mientras la memoria principal se pone al corriente. Con reescritura o copia, el problema es menos severo, pues la escritura a la memoria principal sólo ocurre cuando una línea de caché modificada requiere sustituirse. En el último caso, se necesitan al menos dos accesos a la memoria principal: uno para copiar la antigua línea caché y otra para leer la nueva línea. Un método usado comúnmente para evitar esta penalización por rendimiento debida a la escritura de operaciones es ofrecer a la caché con *buffer de escritura*. Los datos que se escribirán en la memoria principal se colocan en uno de dichos *buffer* y la operación caché continúa sin importar si la escritura a la memoria principal puede ocurrir inmediatamente. En tanto no haya más fallos de caché, la memoria principal no se requiere para actualizarse de inmediato. Las operaciones de escritura se pueden realizar con el uso de ciclos de inactividad del bus hasta que todos los *buffer* de escritura se hayan vaciado. En el caso de otro fallo de caché antes de que todos los *buffer* de escritura se hayan vaciado, se pueden limpiar los *buffer* antes de intentar buscar los datos en la memoria principal u ofrecer un mecanismo especial para buscar en los *buffer* de escritura los datos deseados, y sólo se procede a la memoria principal si la ubicación requerida no está en uno de los *buffer* de escritura.

■ 18.6 Mejoramiento del rendimiento caché

En este capítulo se aprendió que la memoria caché libra la brecha de rapidez entre el CPU y la memoria principal. Las tasas de impacto caché en el rango de 90-98%, e incluso mayores, son comunes, lo cual conduce a alto rendimiento mediante la eliminación de la mayoría de los accesos a memoria principal. Como consecuencia de que las cachés más rápidas son muy costosas, algunas computadoras usan dos o tres niveles de memoria caché, siendo el nivel de caché adicional más grande, más lento y más barato, por byte, que el primero. Una importante decisión de diseño en la introducción de una nueva máquina es el número, capacidades y tipos de memorias caché que se deben usar. Por ejemplo, las cachés divididas producen mayor rendimiento al permitir la concurrencia en acceso a datos e instrucciones. Sin embargo, para una capacidad de caché total dada, las cachés divididas implican una capacidad más pequeña para cada unidad, ello conduce a mayor tasa de fallo que una caché unificada cuando corre un gran programa con poca necesidad de datos o, por el contrario, un programa muy pequeño opera sobre un conjunto de datos muy grande.

Una forma usual de evaluar los méritos relativos de varias alternativas de diseño caché, es la simulación con el uso de conjuntos de datos públicamente disponibles que caractericen accesos a memoria en aplicaciones de interés típicas. Un *rastreo de dirección de memoria* contiene información acerca de la secuencia y temporización de las direcciones de memoria generadas en el curso de la ejecución de conjuntos particulares de programas. Si se proporciona un rastreo de dirección de interés a un simulador

de sistema caché que también se dota con parámetros de diseño para un sistema de caché particular, produce un registro de impactos y fallos en los diversos niveles caché, así como un indicador bastante preciso del rendimiento. Para darse una idea del interjuego de los diversos parámetros, se presentan los resultados de algunos estudios empíricos basados en el enfoque apenas discutido.

En términos generales, las cachés más grandes producen mejor rendimiento. Sin embargo, existe un límite a la validez de esta afirmación. Aquellas son más lentas, de modo que si la capacidad más grande no es necesaria, se puede estar mejor con una caché más pequeña. Además, una caché que encaje en el chip procesador es más rápida debido a alambrados y distancias de comunicación más cortos y a la falta de necesidad de transmisión de señal fuera del chip, que es más bien lento. El requisito de que la caché encaje en el mismo chip con el procesador limita su tamaño, aunque este hecho es menos problemático con el crecimiento en el número de transistores que se pueden colocar en un chip IC. No es raro tener 90% o más de los transistores en un chip CPU asignados a memorias caché.

Aparte del tamaño de caché, usualmente dado en bytes o palabras, otros importantes parámetros de caché son:

1. Ancho de línea 2^W
2. Tamaño de conjunto (asociatividad) 2^S
3. Política de sustitución de línea
4. Política de escritura

Los conflictos relevantes a la elección de estos parámetros, y las negociaciones asociadas, se destacan en el resto de esta sección.

Ancho de línea 2^W : Las líneas más anchas hacen que más datos se lleven a la caché con cada fallo. Si la transferencia de un bloque de datos más grande a la caché se puede lograr a tasa mayor, las líneas de caché más anchas tienen el efecto positivo de propiciar que las palabras que tienen más posibilidad de ingreso en lo futuro (debido a localidad) estén fácilmente accesibles sin fallos posteriores. Asimismo, es posible que el procesador nunca acceda a una parte significativa de una línea de caché muy ancha, ello conduce a desperdicio de espacio caché (que de otro modo podría asignarse a objetos más útiles) y tiempo de transferencia de datos. Debido a estos efectos opuestos, con frecuencia existe un ancho de línea caché óptimo que conduce a mejor rendimiento.

Tamaño de conjunto (asociatividad) 2^S : La negociación aquí es entre la simplicidad y rapidez del mapeo directo y el menor número de fallos de conflicto generados por cachés de conjunto asociativo. Mientras mayor es la asociatividad, menor es el efecto de los fallos de conflicto. Sin embargo, más allá de asociatividad de cuatro u ocho vías, el efecto de rendimiento de mayor asociatividad es despreciable y fácilmente nulificado por los mecanismos de direccionamiento más complejos y lentos. La figura 18.9 muestra resultados experimentales sobre los efectos de la asociatividad sobre el rendimiento. Dado que mayores niveles de asociatividad requieren cabeceras hardware que frenan la caché y también usan alguna área de chip para los mecanismos de comparación y selección necesarios, con frecuencia menos niveles de asociatividad con capacidad más grande (habilitada por el uso del espacio liberado por la remoción de mecanismos de control más complejos para más entradas caché) ofrecen mejor rendimiento global.

Política de sustitución de línea: Por lo general se usa LRU (usado menos recientemente) o alguna aproximación similar para simplificar el proceso de seguir la pista de cuál línea en cada conjunto se debe sustituir a continuación. Para asociatividad de dos vías, la implementación LRU es bastante simple; con cada conjunto se mantiene un solo bit de estado para indicar cuál de las dos líneas en el conjunto tuvo el último acceso. Con cada acceso a una línea, el hardware automáticamente actualiza el bit de estado LRU del conjunto. Conforme la asociatividad aumenta, se vuelve más difícil seguir la huella de los tiempos de uso. En los problemas al final del capítulo se proporcionan algunas ideas

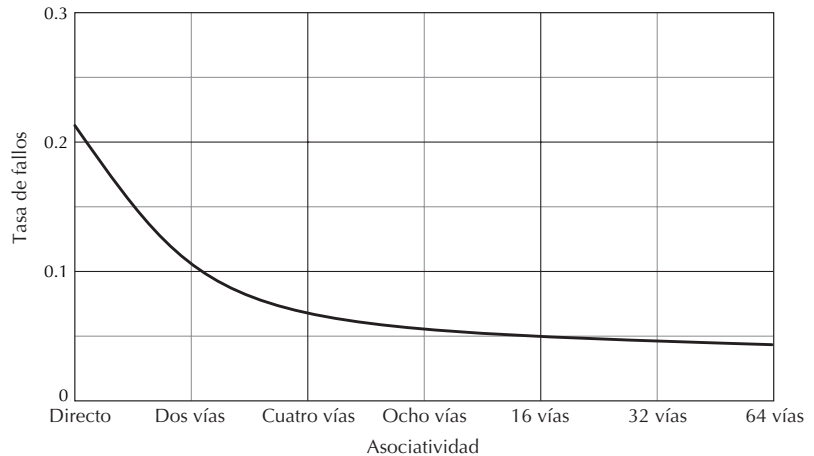


Figura 18.9 Mejora en rendimiento de cachés con asociatividad creciente.

para implementar LRU con cachés de mayor asociatividad. Sorprende que la selección aleatoria de la línea a sustituir funcione tan bien en la práctica.

Política de escritura: Con el esquema de escritura, la memoria principal siempre es consistente con los datos de caché de modo que las líneas de caché se pueden sustituir libremente sin pérdida de información. Con frecuencia es más eficiente una caché de reescritura, pues minimiza el número de accesos a memoria principal. Con las cachés de reescritura se asocia un “bit sucio” con cada línea caché para indicar si ésta se modificó desde que se llevó a la memoria principal. Si una línea no está sucia, su sustitución no crearía problema. De otro modo, la línea caché se debe copiar a la memoria principal antes de que se pueda sustituir. Como ya se mencionó, la escritura real a memoria principal no tiene que ocurrir de inmediato; en vez de ello, las líneas a reescribir se colocan en el *buffer* de escritura y se copian gradualmente según lo permitan la disponibilidad de bus y el ancho de banda de memoria.

Observe que la elección de los parámetros de caché se dictan mediante el patrón dominante y la temporización de los accesos a memoria. Puesto que estas características son significativamente diferentes para corrientes de instrucciones y corrientes de datos, las elecciones óptimas de los parámetros para cachés de instrucción, de datos y unificadas con frecuencia no son las mismas. En consecuencia, no es raro tener diversos grados de asociatividad, o diferentes políticas de sustitución, en unidades de caché múltiples en el mismo sistema.

PROBLEMAS

18.1 Localidad espacial y temporal

Describa un programa de aplicación real cuyos accesos de datos muestren cada uno de los patrones siguientes o argumente que la combinación postulada es imposible.

- Casi ninguna localidad espacial o temporal
- Buena localidad espacial pero virtualmente ninguna localidad temporal
- Buena localidad temporal pero muy poca o ninguna localidad espacial
- Buenas localidades espacial y temporal

18.2 Rendimiento caché de dos niveles

Un procesador con dos niveles de caché tiene un CPI de 1 cuando no hay fallo de caché nivel 1. En el nivel 1, la tasa de impacto es 95% y un fallo incurre en penalización de diez ciclos. Para la caché de dos niveles como un todo, la tasa de impacto es 98% (ello significa que 2% del tiempo se debe acceder a la memoria principal) y la penalización por fallo es de 60 ciclos.

- ¿Cuál es el CPI efectivo después de que los fallos de caché se factorizan?

- b) Si en lugar de este sistema caché de dos niveles se usara una caché de un solo nivel, ¿qué tasa de impacto y penalización por fallo se necesitarían para proporcionar el mismo rendimiento?
- e) Conjunto asociativo de dos vías, líneas de cuatro palabras, capacidad de dos conjuntos, sustitución LRU.
- f) Conjunto asociativo de cuatro palabras, líneas de dos palabras, capacidad de dos conjuntos, sustitución LRU.

18.3 Rendimiento de caché de dos niveles

Un sistema de cómputo usa dos niveles de caché L1 y L2. Al nivel L1 se ingresa en un ciclo de reloj y suministra los datos en caso de un impacto L1. Para un fallo L1, que ocurre durante 3% del tiempo, se consulta L2. Un impacto L2 incurre en penalización de diez ciclos de reloj mientras que un fallo L2 implica penalización de 100 ciclos.

- a) Si supone implementación encauzada con un CPI de 1 cuando no hay fallo de caché (es decir, si se ignoran las dependencias de datos y control), calcule el CPI efectivo si la tasa de fallo local de L2 es 25%.
- b) Si se tuviese que modelar el sistema caché de dos niveles como una sola caché, ¿qué tasa de fallo y penalización por fallo se deben usar?
- c) Cambiar el esquema de mapeo de L2 de directo a conjunto asociativo de dos vías, puede mejorar su tasa de fallo local a 22% mientras aumenta su penalización por impacto a 11 ciclos de reloj, debido al esquema de acceso más complejo. Si ignora cuestiones de costo, ¿el cambio será buena idea?

18.4 Impactos y fallos de memoria caché

La siguiente secuencia de números representa las direcciones de memoria en una memoria principal de 64 palabras: 0, 1, 2, 3, 4, 15, 14, 13, 12, 11, 10, 9, 0, 1, 2, 3, 4, 56, 28, 32, 15, 14, 13, 12, 0, 1, 2, 3. Clasifique cada uno de los accesos como un impacto caché, fallo obligatorio, de capacidad o de conflicto, dados los siguientes parámetros de caché. En cada caso proporcione los contenidos finales de caché.

- a) Mapeo directo, líneas de cuatro palabras, capacidad de cuatro líneas.
- b) Mapeo directo, líneas de dos palabras, capacidad de cuatro líneas.
- c) Mapeo directo, líneas de cuatro palabras, capacidad de dos líneas.
- d) Conjunto asociativo de dos vías, líneas de dos palabras, capacidad de cuatro conjuntos, sustitución LRU.

18.5 Impactos y fallos de memoria caché

Un programa tiene un ciclo de nueve instrucciones que se ejecuta muchas veces. Sólo la última instrucción del ciclo constituye una bifurcación cuyo destino es la primera instrucción del ciclo en la dirección de memoria 5678. La primera instrucción del ciclo lee el contenido de una localidad de memoria diferente cada vez, comenzando con la dirección de memoria 8760 y va de uno en uno en cada nueva iteración. Determine la tasa de impacto de memoria caché, como un todo y por separado para instrucciones y datos, de acuerdo con los siguientes parámetros de caché.

- a) Mapeo directo unificado, líneas de cuatro palabras, capacidad de cuatro líneas
- b) Conjunto asociativo de dos vías unificado, líneas de dos palabras, capacidad de cuatro conjuntos, sustitución LRU
- c) Cachés divididas de mapeo directo, cada una tiene líneas de cuatro palabras y capacidad de dos líneas
- d) Cachés divididas de conjunto asociativo de dos vías, cada una tiene líneas de dos palabras, capacidad de dos conjuntos, sustitución LRU

18.6 Diseño de memoria caché

Caracterice el rastreo de dirección producido por la ejecución del ciclo definido en el problema 18.5, si supone un número infinito de iteraciones de ciclo y tamaños de caché que son potencias de 2. Encuentre la caché más pequeña posible (o la capacidad de caché total más pequeña en el caso de cachés divididas) para lograr una tasa de fallo que no supere 5% con cada una de las siguientes restricciones sobre la organización de caché. Tiene libertad de elegir cualquier parámetro que no se especifique explícitamente.

- a) Caché unificada de mapeo directo.
- b) Caché unificada de conjunto asociativo de dos vías.
- c) Cachés divididas de mapeo directo.
- d) Cachés divididas de conjunto asociativo de dos vías.

18.7 Diseño de memoria caché

- a) Considere la memoria caché que se bosqueja en la figura 18.4. Para una capacidad de memoria principal de 2^x palabras, determine el número total de bits en el arreglo caché y derive la aportación de los bits con la negación de datos como una cabecera porcentual relativa a los bits de datos reales.
- b) Repita la parte a) para la memoria caché que se muestra en la figura 18.6.

18.8 Diseño de memoria caché

Un sistema de cómputo tiene 4 GB de memoria principal direccionable por byte y una memoria caché unificada de 256 KB con bloques de 32 bytes.

- a) Dibuje un diagrama que muestre cada uno de los componentes de una dirección de memoria principal (es decir, cuántos bits para tag, índice de conjunto y *offset* de byte) para una caché de conjunto asociativo de cuatro vías.
- b) Dibuje un diagrama que muestre los circuitos de comparación de tag, generación de la señal de fallo de caché y la salida de datos para la caché.
- c) El rendimiento del sistema de cómputo con arquitectura caché de conjunto asociativo de cuatro vías resulta insatisfactorio. Se consideran dos opciones de rediseño que implican casi los mismos diseños adicional y costos de producción. La opción A consiste en aumentar el tamaño de la caché a 512 KB. La opción B representa aumentar la asociatividad de la caché de 256 KB a 16 vías. A su juicio, ¿cuál opción tiene más probabilidad de resultar en mayor rendimiento global y por qué?

18.9 Dirección de caché en una memoria direccionable por byte

Una dirección para una memoria direccionable por byte presentada a la unidad caché se divide así: tag

de 13 bits, índice de línea de 14 bits, *offset* de byte de cinco bits.

- a) ¿Cuál es el tamaño caché en bytes?
- b) ¿Cuál es el esquema de mapeo caché?
- c) Para un byte dado en caché, ¿cuántos bytes diferentes en la memoria principal de 2^{32} bytes pueden ocuparla?

18.10 Relaciones entre parámetros de caché

Para cada una de las memorias caché parcialmente especificadas en la tabla siguiente, encuentre los valores para los parámetros perdidos donde sea posible o explique por qué no se pueden deducir de manera única. Suponga memorias principal y caché direccionables por palabra.

18.11 Relaciones entre parámetros de caché

Si usa a para bits de dirección, c_w para palabras en caché, l_w para palabras por línea, s_l para líneas por conjunto, c_s para conjuntos en caché, t para bits de tag, i para bits de índice de conjunto y o para bits de *offset*, escriba ecuaciones que relacionen los parámetros mencionados en el problema 18.10, de modo que cuando se asignen valores a un subconjunto de parámetros se obtengan valores únicos, o un conjunto de valores factible, para otros parámetros.

18.12 Negociaciones en diseño de caché

Se consideran tres opciones de diseño para una memoria caché de mapeo directo de 16 palabras. Dichas opciones C1, C2 y C4 corresponden a usar líneas caché de una palabra, dos palabras y cuatro palabras, respectivamente. Las penalizaciones por fallo para las opciones C1, C2 y C4 son seis, siete y nueve ciclos de reloj, respectivamente. Proporcione rastreos de dirección de memoria relativamente cortos que conduzcan a cada uno

Parte del problema	Bits de dirección	Palabras en caché	Palabras por línea	Líneas por conjunto	Conjuntos en caché	Bits de tag	Bits de índice de conjunto	Bits de <i>offset</i>
a	16	1024	4	1				
b	24	4096	8	2				
c					64	12	6	0
d					128	23	7	2
e			16	8			5	3
f	12	256			32	6		

de los resultados siguientes o muestre que el resultado es imposible.

- a) C2 tiene más fallos que C1.
- b) C4 tiene más fallos que C2.
- c) C1 tiene más fallos que C4.
- d) C2 tiene menos fallos que C1, pero desperdicia más ciclos en los fallos de caché.
- e) C4 tiene menos fallos que C2, pero desperdicia más ciclos en fallos de caché.

18.13 Programación atenta al caché

Un problema que se presenta con las cachés directas e incluso con los conjuntos asociativos es el *thrashing* (hiperpaginación), que se define como fallos de conflicto excesivos como consecuencia de la naturaleza de patrones de dirección de memoria particulares. Considere una caché de mapeo directo que contiene dos líneas de cuatro palabras usadas en el curso del cálculo del producto interno de dos vectores de longitud 8 en una forma que involucra leer A_i y B_j , multiplicarlos y sumar el producto a un total corriente en un registro.

- a) Demuestre que el *thrashing* ocurre cuando las direcciones de inicio de A y B son múltiplos de 4.
- b) Sugiera formas de evitar el *thrashing* en este ejemplo.
- c) Para este ejemplo particular, ¿el *thrashing* sería posible si la caché fuese de conjunto asociativo de dos vías?

18.14 Comportamiento de caché durante la transposición de matriz

La transposición de matriz constituye una operación importante en muchos procesamiento de señales y cálculos científicos. Considere la transposición de la matriz A de 4×4 , que se almacena en orden de renglón mayor comenzando en la dirección 0 y cuyo resultado se coloca en B , almacenado en orden de renglón mayor comenzando en la dirección 16. Se usan dos ciclos anidados (índices i y j), y se copia $A_{i,j}$ en $B_{j,i}$ dentro del ciclo interior j . Suponga unidades de memoria caché y principal direccionables por palabra. Observe que a cada elemento de matriz se accede exactamente una vez. Dibuje dos tablas 4×4 que representen los elementos de matriz y coloque “H” (para impacto) o “M” (para fallo) en cada entrada para las siguientes organizaciones de una caché de ocho palabras.

- a) Mapeo directo, líneas de dos palabras.
- b) Mapeo directo, líneas de cuatro palabras.
- c) Conjunto asociativo de dos vías, líneas de una palabra, política de sustitución LRU.
- d) Conjunto asociativo de dos vías, líneas de dos palabras, política de sustitución LRU.

18.15 Implementación de política LRU

Una forma de implementar la política de sustitución LRU para una caché de conjunto asociativo de cuatro vías consiste en almacenar seis bits de estado por cada conjunto: el bit $b_{i,j}$, $0 \leq i < j \leq 3$, se fija a 1 si la línea i se accedió más recientemente que a la línea j en el conjunto. Proporcione detalles de implementación para este esquema, incluso el circuito lógico que actualiza los seis bits de estado y seleccione la línea a sustituir.

18.16 Caché para evitar cálculos repetidos

En algunas aplicaciones (notablemente en multimedia) ciertas instrucciones se ejecutan repetidamente con los mismos argumentos, ello produce resultados similares. La mayoría de las instrucciones, como la suma o la multiplicación se ejecutan tan rápido que repetir las quizá es más prudente que intentar detectar sus repeticiones por usar resultados calculados. Otras instrucciones, como *load* o *store* pueden producir resultados diferentes aun cuando las direcciones de memoria y registro involucrados sean las mismas. Sin embargo, para el caso de la división, que tarda 20-40 ciclos de reloj en ejecutarse en muchos microprocesadores modernos, se puede lograr ahorros significativos si los resultados de las divisiones previamente realizadas se almacenan en un *buffer*, que se consulta en un ciclo de reloj para ver si el resultado necesario está disponible ahí. Si ocurre un impacto, el cociente calculado se envía al registro apropiado. Sólo en el caso de un fallo, por ejemplo en 10-20% de los casos, los operandos se envían a la unidad divisora para su cálculo. A este método se le denomina *memoing* y al *buffer* que contiene los argumentos y resultados para las recientes instrucciones de división como *division memo table* (DMT).

- a) Dibuje un diagrama de bloques para una DMT de mapeo directo de 64 entradas y muestre cómo funciona; esto es, muestre qué se debe almacenar en ella, cómo se consulta y cómo se obtiene y usa la señal de fallo/impacto.

- b) Repita la parte a) para una DMT de conjunto asociativo de cuatro vías y resalte las ventajas e inconvenientes del esquema modificado, si lo hubiera.
- c) Si supone una instrucción divide de 22 ciclos (un ciclo preparación/decodificación, ciclo de búsqueda en DMT y 20 ciclos de ejecución si hubiera fallo DMT), exprese el número promedio de ciclos de reloj con la operación de dividir y el factor s de aceleración de auxilio de división como función de la tasa de impacto h en la DMT. ¿Qué tasa de impacto DMT h se necesita para una aceleración de división de al menos 3?
- d) Dibuje y explique otro diagrama hardware para el esquema *memoing* que no involucre la penalización de un ciclo en caso de fallos en la DMT (es decir, la división tarda 21 ciclos en el peor caso).
- e) Repita la parte c) para la implementación de la parte d). ¿Qué fracción del tiempo de ejecución se debe emplear en la división, si una aceleración de la división por un factor promedio de 3 produce una aceleración global de 5%?

18.17 Caché de rastreo para instrucciones

Una caché de rastreo conserva instrucciones que no están en orden de dirección sino de ejecución dinámica. Esto último aumenta el ancho de banda de lectura (*fetch*) de instrucción porque un bloque de instrucciones leído (*fetch*) quizá se ejecute en su totalidad. Estudie los problemas y ventajas de diseño para una caché de rastreo y escriba un informe de dos páginas acerca de ello [Rote99].

REFERENCIAS Y LECTURAS SUGERIDAS

- [Crag96] Cragon, H. G., *Memory Systems and Pipelined Processors*, Jones and Bartlett, 1996.
- [Lipt68] Liptay, J. S., "Structural Aspects of the System/360 Model 85, part II: The Cache", *IBM Systems J.*, vol. 7, núm. 1, pp. 15-21, 1968.
- [Pat98] Patterson, D. A. y J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann, 2a. ed., 1998.
- [Rote99] Rotenberg, E., S. Bennett y J. E. Smith, "A Trace Cache Microarchitecture and Evaluation", *IEEE Trans. Computers*, vol. 48, núm. 2, pp. 111-120, febrero de 1999.
- [Shri98] Shriver, B. y B. Smith, *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*, IEEE Computer Society Press, 1998, cap. 5, pp. 427-461.
- [Smit82] Smith, A. J., "Cache Memories", *ACM Computing Surveys*, vol. 14, núm. 3, pp. 473-530, septiembre de 1982.

CONCEPTOS DE MEMORIA MASIVA

“Creo que Silicon Valley (Valle de silicio) es un nombre equivocado. Si revisas la tremenda cantidad en dólares ganados como consecuencia de la venta de productos durante la última década, te darás cuenta de que ha habido más ingresos por la venta de discos magnéticos que por el silicio. Debieran renombrar el lugar como Iron Oxide Valley (Valle del óxido de hierro).”

Al Hoagland, 1982

“Un memex constituye un dispositivo en el que un individuo almacena todos sus libros, discos y comunicaciones, y está mecanizado de modo que se pueda consultar con excesiva rapidez y flexibilidad. Representa un íntimo complemento agrandado de su memoria.”

Vannevar Busch, Cómo podemos pensar, 1945

TEMAS DEL CAPÍTULO

- 19.1** Fundamentos de memoria de disco
- 19.2** Organización de datos en disco
- 17.3** Rendimiento de disco
- 19.4** *Caching* de disco
- 19.5** Arreglos de discos y RAID
- 17.6** Otros tipos de memoria masiva

El tamaño de las memorias principales en una computadora moderna supera la imaginación de los primeros programadores de computadoras, aunque hoy se estima muy pequeña para contener todos los datos y programas de interés de un usuario típico. Además, la volatilidad de las memorias semiconductores, esto es, el borrado de datos en el evento de pérdida de energía, necesitaría una forma de almacenamiento de respaldo (*backup*), incluso si el tamaño no fuese un problema. Con los años, las memorias de disco han asumido este rol dual de almacenamiento extendido y respaldo. Las mejoras en la capacidad y densidad de almacenamiento y las reducciones en el costo de las memorias de disco, han sido tan fenomenales como los avances en los circuitos integrados. En este capítulo se estudiarán la organización y rendimiento de la memoria de disco y se verá cómo el caché de disco y la organización redundante de arreglos de discos múltiples, conduce a mejoras en la rapidez y la confiabilidad. También se revisan algunas otras tecnologías de almacenamiento masiva.

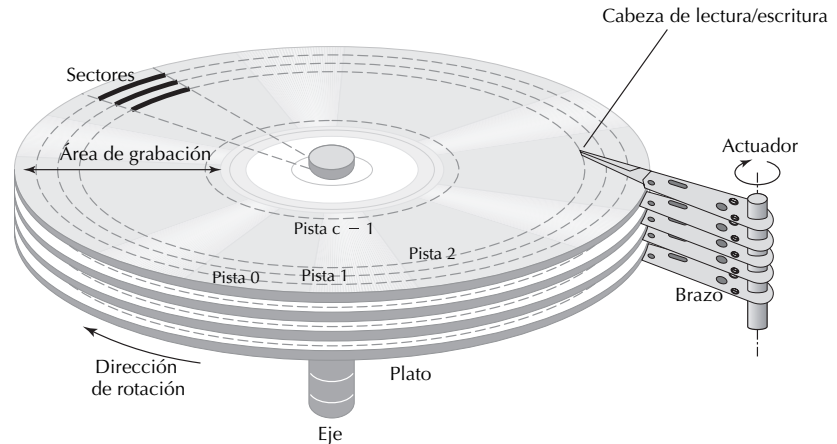


Figura 19.1 Elementos de memoria de disco y términos clave.

■ 19.1 Fundamentos de memoria de disco

La mayor parte de este capítulo se dedica a una discusión de las memorias de disco duro de grabación magnética a las que se hará referencia como “memoria de disco” para abreviar. Los discos flexibles (*floppy disk*), discos ópticos (CD-ROM, CD-RW, DVD-ROM, DVD-RW) y otras variantes de discos que tienen organizaciones y principios operativos similares, pero difieren en sus tecnologías de grabación y acceso, áreas de aplicación y criterios de rendimiento. Estas variantes se discuten en la sección 19.6, junto con algunas otras opciones de almacenamiento masivo.

Los discos duros modernos son maravillas de diseño eléctrico y mecánico. Es más bien curioso que tales discos se usen en absoluto. Parece que las memorias electrónicas deben sustituir la memoria de disco relativamente lenta a corto plazo, como se predijo más de una vez en décadas pasadas. Sin embargo, la mejora fenomenal en la densidad de grabación de datos, el rápido declive en costos y la muy alta confiabilidad de los modernos sistemas de disco ha llevado a su uso continuo en virtualmente todas las computadoras, comenzando con laptop y de escritorio.

La figura 19.1 muestra una configuración típica de memoria de disco y la terminología asociada con su diseño y uso. Hay de 1-12 *platos* (*platters*) montados en un *eje* (*spindle*) que rota con rapidez de 3 600 a 10 000 revoluciones, o más, por minuto. Los datos se graban en ambas superficies de cada plato a lo largo de *pistas* (*tracks*) circulares. Cada pista se divide en muchos *sectores*, con un sector que representa la unidad de transferencia de datos hacia y desde el disco. La densidad de grabación es función de la *densidad de pistas* (pistas por centímetro o pulgada) y la *densidad de bit lineal* a lo largo de la pista (bits por centímetro o pulgada). En el año 2000 la *densidad de grabación de área* de los discos comerciales baratos oscilaba de 1-3 Gb/cm². Los primeros discos de computadora tenían diámetros de hasta 50 cm, pero los discos modernos casi no salen del rango de 1.0-3.5 pulgadas (2.5-9 cm) de diámetro.

El *área de grabación* en cada superficie no se extiende por todo el centro del plato porque las pistas muy cortas cerca del medio no se pueden utilizar eficientemente. Aun así, las pistas interiores son un factor de 2 o más cortas que las pistas exteriores. Tener el mismo número de sectores en todas las pistas limitaría la pista (y la capacidad del disco) en lo que es posible grabar en las pistas interiores cortas. Por esta razón, los discos modernos ponen más sectores en las pistas exteriores. Los bits grabados en cada sector incluyen un número de sector al principio, seguido de una brecha que permite que el número de sector se procese y note por la lógica de la cabeza de lectura/escritura, los datos del sector y la información de detección/corrección de error. También hay una brecha entre sectores adyacentes.

Estas brechas, número de sector y cabecera de codificación de error, más pistas de repuesto (*spare*), se usan con frecuencia para permitir la “reparación” de pistas malas descubiertas en el curso de la operación de memoria de disco, de modo que la *capacidad formateada* de un disco resulta menor que su capacidad bruta basada en la densidad de grabación de datos.

Un *actuador* puede mover los brazos que sostienen las cabezas de lectura/escritura, de las que se tienen tantas como superficies de grabación hay, para alinearlas con un *cilindro* deseado, que consta de pistas con el mismo diámetro en diferentes superficies de grabación. La lectura de datos muy cercanamente espaciados en el disco requiere que la cabeza viaje muy cerca a la superficie del disco (dentro de una fracción de micrómetro). Mediante un delgado colchón de aire se evita que las cabezas choquen con la superficie. Observe que incluso la partícula de polvo más pequeña puede chocar en la superficie. Tales *choques de la cabeza* dañan las partes mecánicas y destruyen gran cantidad de datos en el disco. Para evitar estos eventos indeseables, los discos duros se sellan en empaques herméticos.

El tiempo de acceso a los datos en un sector deseado en el disco consta de tres componentes:

1. *Tiempo de búsqueda*, o tiempo para alinear las cabezas con el cilindro que contiene la pista en la que reside el sector.
2. *Latencia rotacional*, o tiempo para que el disco rote hasta que el comienzo de los datos de sector llegue bajo la cabeza de lectura/escritura.
3. *Tiempo de transferencia de datos*, que consiste en el tiempo para que el sector pase bajo la cabeza que lee los bits al vuelo.

La tabla 19.1 contiene datos acerca de las características clave de tres discos modernos con diferentes parámetros físicos y dominios de aplicación pretendidos. La capacidad de una memoria de disco se relaciona con alguno de estos parámetros así:

$$\text{Capacidad de disco} = \text{superficies} \times \text{pistas/superficie} \times \text{sectores/pista} \times \text{bytes/sector}$$

Con frecuencia el número de superficies de grabación resulta el doble del número de platos. Por esta razón, los discos con mayor capacidad son de la variedad multiplatos. Tanto el diámetro del disco como la densidad de pistas afectan el número de pistas por superficie. Los bytes por pista, que es el producto de los primeros dos términos, son proporcionales a la longitud de pista (y al diámetro de pista) y a la densidad de bit lineal. Cuando todas las pistas no contienen el mismo número de sectores, en la ecuación de capacidad de disco se usa el número promedio de sectores por pista.

Ejemplo 19.1: Parámetros de memoria de disco Calcule la capacidad de una unidad de disco de dos platos con 18 000 cilindros, un promedio de 520 sectores por pista y un tamaño de sector de 512 B.

Solución: Con dos platos, hay cuatro superficies de grabación. Por tanto, la máxima capacidad bruta del disco es $4 \times 18000 \times 520 \times 512 \text{ B} = 1.917 \times 10^{10} \text{ B} = 1.917 \times 10^{10}/2^{30} \text{ GB} = 17.85 \text{ GB}$. Si se permite 10% para cabecera o desperdicio de capacidad para brechas, números de sector y codificación para verificación de redundancia cíclica (CRC), se llega a una capacidad formateada de casi 16 GB.

Con base en las observaciones anteriores, se deduce que los *discos duros* retienen más datos y son más rápidos que los *flexibles* porque tienen:

- Diámetros más grandes (aunque ya no se fabrican discos anchos).
- Giro más rápido.
- Grabación a densidades más altas (debido a control más preciso).
- Múltiples platos.

TABLA 19.1 Atributos clave de tres discos magnéticos representativos, desde la capacidad más alta hasta el tamaño físico más pequeño (*circa* principios 2003).

	Fabricante	Seagate	Hitachi	IBM
Identidad del disco	Serie	Barracuda	DK23DA	Microdrive
	Modelo	180 SCSI	ATA-5	1 GB
	Número modelo	ST1181677LW	40	DSCM-11000
	Dominio aplicación común	Servidor/escritorio	Laptop	Dispositivo de bolsillo
Atributos de almacenamiento	Capacidad formateada, GB	180	40	1
	Superficies grabación	24	4	2
	Cilindros	24 247	33 067	7 167
	Tamaño sector, B	512	512	512
	Sectores/pistas prom	604	591	140
	Máx densidad grabación, Kb/cm	198	236	171
	Máx densidad pista, pistas/cm	12 280	21 700	13 780
	Máx densidad por área, Gb/cm ²	2.4	5.1	2.4
	Tamaño <i>buffer</i> , MB	16	2	1/8
	Mín tiempo búsqueda, ms	1	3	1
Atributos de acceso	Tiempo búsqueda prom, ms	8	13	12
	Máx tiempo búsqueda, ms	17	25	19
	Mín tasa datos interna, MB/s	25.3	18.7	2.6
	Máx tasa datos interna, MB/s	47.0	34.7	4.2
	Tasa datos externa, MB/s	160	100	13
Atributos físicos	Diámetro, pulgadas	3.5	2.5	1
	Platos	12	2	1
	Rapidez rotación, rpm	7 200	4 200	3 600
	Empaque $L \times A \times F$, cm	$10.2 \times 4.1 \times 14.6$	$7.0 \times 1.0 \times 10.0$	$4.3 \times 0.5 \times 3.6$
	Peso empaque, kg	1.04	0.10	0.04
	Potencia operación típica, W	14.1	2.3	0.8
	Potencia inactiva, W	10.3	0.7	0.5

Los discos ópticos usan un haz láser y su reflejo en una sola superficie de grabación, en lugar de magnetización y detección de campo magnético, para escribir y leer datos. El control más preciso del mecanismo luminoso permite grabación más densa, así como mayor capacidad. En la sección 19.6 se discuten con más detalle este tipo de discos.

■ **19.2 Organización de datos en disco**

Los bits de datos en una pista aparecen como pequeñas regiones del recubrimiento magnético en la superficie del disco que se magnetizan en direcciones opuestas para grabar 0 o 1 (figura 19.2). Conforme las áreas asociadas con diferentes bits pasan bajo el mecanismo lector, se percibe la dirección de magnetización y se deduce el valor de bit. La grabación ocurre cuando el mecanismo de escritura fuerza la magnetización en una dirección deseada al pasar una corriente a través de una bobina unida a la cabeza. Este esquema se denomina *grabación horizontal*. También es posible usar *grabación vertical* en la que la dirección de magnetización es perpendicular a la superficie de grabación. La grabación vertical permite que los bits se coloquen más juntos, pero requiere medios de grabación magnética diseñados especialmente junto con mecanismos de lectura y escritura más complejos, no es de uso común.

Los 0 y 1 que corresponden a las direcciones de magnetización dentro de celdas pequeñas en la superficie del disco, como se muestra en la figura 19.2, usualmente no representan de manera directa valores de bit de datos porque se usan técnicas de codificación especiales para maximizar la densidad

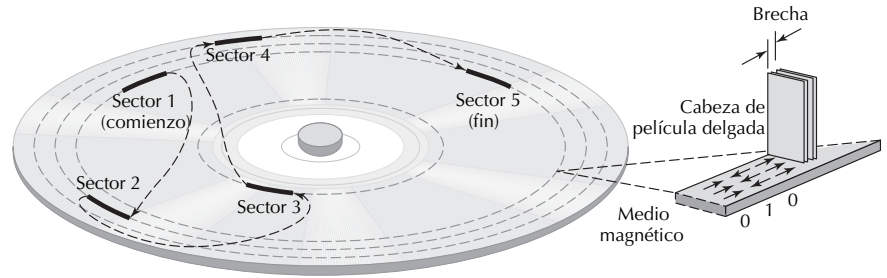


Figura 19.2 Grabación magnética a lo largo de las pistas y la cabeza de lectura/escritura.

de almacenamiento y hacer más probable el funcionamiento correcto de los mecanismos de lectura y escritura. Por ejemplo, en lugar de dejar que los valores de bit de datos dicten la dirección de magnetización, se puede magnetizar con base en si hay un cambio en valor del bit de una celda a la siguiente. Este tipo de codificación *no regresa a cero* (NRZ, por sus siglas en inglés), que permite la duplicación de la capacidad de grabación de datos en relación con la codificación *regresa a cero* (RTZ, por sus siglas en inglés), representó la primera técnica de codificación usada ampliamente. La discusión de tales códigos no pertenece al ámbito de este libro. La visión simplificada que la cadena “0 1 0” muestra en la figura 19.2, y que representa los valores de tres bits de datos consecutivos en la pista del disco, es suficiente para los propósitos del texto.

Además de las complejidades de los códigos usados para la grabación magnética, cada sector de datos está precedido por un número de sector grabado y seguido por una *verificación de redundancia cíclica*, que permite la detección o corrección de ciertas anomalías de grabación y errores de lectura/escritura. Esto es esencial porque, a densidades de grabación muy altas, la magnetización y detección de errores son inevitables. Asimismo, hay brechas entre sectores y brechas dentro de sectores para separar los diversos componentes de sector para que se complete el procesamiento de una parte antes de que llegue la siguiente. De nuevo, la visión simplificada de que los sectores se siguen mutuamente, sin cabecera de grabación ni brecha, es adecuada para el propósito del texto.

La unidad de transferencia de datos hacia/desde el disco es un sector que usualmente contiene de 512 a unos cuantos miles de bytes. La dirección de un sector en el disco consta de tres componentes:

Dirección disco = cilindro #,	pista #,	Sector #
17–31 bits	10–16 bits,	1–5 bits, 6–10 bits

Con un tamaño de sector de $512 = 2^9$ B, esto último representa una capacidad de disco de:

$$\text{Capacidad de disco} = 2^{13 \pm 3} \times 2^{3 \pm 2} \times 2^{8 \pm 2} \times 2^9 = 2^{33 \pm 7} \text{ B} = 0.06 - 1024 \text{ GB}$$

Desde luego, los discos actuales no permiten que estos parámetros estén simultáneamente en sus valores máximos. El número de cilindros se proporciona al mecanismo actuador para alinear las cabezas de lectura/escritura con el cilindro deseado. El número de pista selecciona una de las cabezas de lectura/escritura (o la superficie de grabación asociada). Finalmente, el número de sector se compara contra los índices de sector grabados conforme pasan bajo la cabeza de lectura/escritura seleccionada, y una identidad indica que el sector deseado llegó bajo la cabeza. Entonces la cabeza lee los datos del sector actual y su información de detección/corrección de error asociada.

Los sectores en un disco son independientes, cualquier colección de sectores se puede usar para contener varias partes de un archivo u otra estructura de datos. Por ejemplo, la figura 19.2 muestra un archivo compuesto de cinco sectores que están dispersos en diferentes pistas. Las direcciones de estos sectores se pondrán en un directorio archivo de modo que cualquier pieza del archivo se recupere cuando se necesite; de manera alterna, sólo el primer sector del archivo se puede poner en el directorio.

rio, con una liga proporcionada dentro de cada sector hacia el siguiente sector. Desde luego, dado que las piezas de un archivo quizá se accedan en cercana proximidad una de otra, tiene sentido intentar asignar sectores que reducirían la cantidad de movimientos de cabeza (tiempo de búsqueda) y latencia rotacional en el curso de acceso a archivo.

Ejemplo 19.2: Elaboración de una copia de respaldo de todo un disco Un disco con $t = 100\,000$ pistas (el número total en todas las superficies de grabación) rota a 7 200 rpm, su tiempo de búsqueda de pista a pista es 1 ms. ¿Cuál es la cantidad mínima de tiempo necesario para hacer una copia de respaldo de todos los contenidos del disco? Suponga que el disco está lleno y que el dispositivo que hace el respaldo acepta datos a la tasa proporcionada por el disco.

Solución: El disco da 120 revoluciones por segundo, o una evolución en $1/120$ s. Ignore la latencia rotacional de un tiempo al arranque, que dura en promedio 4.17 ms. Para copiar todos los contenidos del disco se necesitan $t - 1 = 99\,999$ búsquedas de pista a pista y 100 000 revoluciones completas para transferencia de datos. Por tanto, el tiempo total es $(t - 1)/10^3 + t/120 \cong 933$ s $\cong 15.5$ min. Este cálculo supone que los sectores se pueden leer en orden de su aparición física en la pista. En otras palabras, la lectura de datos de un sector se descarga inmediatamente de modo que el siguiente sector se lea sin retardo. Dentro de poco se verá que éste puede no ser el caso.

En virtud de que el tiempo de búsqueda es mucho menor cuando se va de un cilindro a otro adyacente (en oposición a uno distante), los elementos de datos a los que usualmente se accede juntos se colocarán en el mismo o en cilindros adyacentes. Sin embargo, los datos a los que se accede consecutivamente no se les debe asignar sectores sucesivos en la misma pista. La razón es que con frecuencia se requiere alguna cantidad de procesamiento en un sector antes de acceder al siguiente. Por tanto, colocar la siguiente parte de datos en el sector inmediato conduce a la posibilidad de perder la oportunidad de leerlo y esperar durante casi una rotación completa del disco. Por esta razón, los *números de sector lógico* consecutivos se asignan a *sectores físicos* que están separados por algunos sectores intermedios. De esta forma, colocar las piezas de un archivo en sectores lógicos consecutivos no conduciría al problema descrito. Adicionalmente, los números de sector lógico en pistas adyacentes se pueden sesgar. Suponga que se acaba de completar la lectura del último sector lógico en una pista y ahora se debe mover al primer sector lógico en la pista siguiente para obtener la siguiente pieza del archivo. Es posible que la cabeza tarde un par de milisegundos en completar el movimiento de una pista a la siguiente, tiempo durante el cual el disco pudo haber rotado en un ángulo bastante grande. La figura 19.3 muestra un posible esquema de numeración para sectores en varias pistas adyacentes con base en las observaciones anteriores.

Por fortuna el usuario no necesita preocuparse por el esquema real de colocación de datos en la superficie del disco. El controlador del disco mantiene la información de mapeo requerida para interpretar un número de sector secuencial simple, presentado por el sistema operativo, en una dirección de disco físico que identifique la superficie, pista y sector. Entonces el controlador planea y realiza todas

	0	16	32	48	1	17	33	49	2		Pista i
	30	46	62	15	31	47	0	16	32		Pista i + 1
	60	13	29	45	61	14	30	46	62		Pista i + 2
	27	43	59	12	28	44	60	13	29		Pista i + 3

Figura 19.3 Numeración lógica de sectores en varias pistas adyacentes.

las actividades requeridas para recolectar los datos de sector en un *buffer* local antes de transferirla a un área designada en la memoria principal.

19.3 Rendimiento de disco

El rendimiento del disco se relaciona con la *latencia de acceso* y la *tasa de transferencia de datos*. La latencia de acceso constituye la suma de tiempo de búsqueda de cilindro (o *tiempo de búsqueda*) y la *latencia rotacional*, tiempo necesario para que el sector de interés llegue bajo la cabeza de lectura/escritura. Por tanto:

$$\text{Latencia de acceso de disco} = \text{tiempo de búsqueda} + \text{latencia rotacional}$$

El tiempo de búsqueda depende de cuánto tenga que viajar la cabeza desde su posición actual al cilindro destino. Dado que esto implica un movimiento mecánico, que consiste de una fase de aceleración, un movimiento uniforme y una fase de desaceleración o freno, se puede modelar el tiempo de búsqueda para moverse por c cilindros del modo siguiente, donde α , β y γ son constantes:

$$\text{Tiempo de búsqueda} = \alpha + \beta(c - 1) + \gamma\sqrt{c - 1}$$

El término lineal $\beta(c - 1)$, correspondiente a la fase de movimiento uniforme, más bien es una adición reciente a la ecuación de tiempo de búsqueda; los discos antiguos no tienen suficientes pistas o aceleración suficiente alta para iniciar movimiento uniforme.

La latencia rotacional constituye una función respecto de dónde se ubica el sector deseado en la pista. En el mejor caso, la cabeza está alineada con la pista justo cuando llega el sector deseado. En el peor caso, la cabeza acaba de perder el sector y debe esperar durante casi una revolución completa. De ese modo, en promedio, la latencia rotacional es igual al tiempo para media revolución:

$$\text{Latencia rotacional promedio} = (30/\text{rpm}) \text{ s} = (30\,000/\text{rpm}) \text{ ms}$$

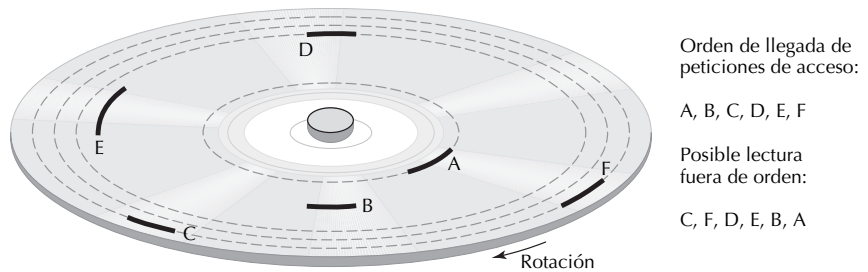
En consecuencia, para una rapidez de rotación de 10000 rpm, la latencia rotacional promedio es 3 ms y su rango es 0-6 ms.

La tasa de transferencia de datos se relaciona con la rapidez rotacional del disco y la densidad de bit lineal a lo largo de la pista. Por ejemplo, suponga que una pista contiene casi 2 Mb de datos y el disco rota a 10000 rpm. Entonces, cada minuto, 2×10^{10} bits pasan bajo la cabeza. En virtud de que los bits se leen con rapidez conforme pasan bajo la cabeza, ello significa una tasa de transferencia de datos promedio de casi $333 \text{ Mb/s} = 42 \text{ MB/s}$. La cabecera inducida por brechas, números de sector y codificaciones CRC propician que la tasa de transferencia pico sea un poco más alta que la tasa de transferencia promedio calculada.

Los intentos por eliminar el tiempo de búsqueda, que de acuerdo con la tabla 9.1, constituyen una fracción significativa de la latencia de acceso al disco, no han tenido éxito. Los *discos de cabeza por pista*, que tienen una cabeza permanentemente alineada con cada pista de cada superficie de registro (usualmente 1-2), no tuvieron éxito debido a su complejidad y mercado limitado, que hicieron los costos de diseño y fabricación inaceptablemente altos. Como consecuencia de que la rapidez de rotación afecta tanto la latencia rotacional como la tasa de transferencia de datos, los fabricantes de discos están motivados por aumentarla en la medida posible. Más allá de esto último, los esfuerzos por cortar la latencia rotacional promedio por un factor de 2-4, mediante la colocación de múltiples conjuntos de cabezas de lectura/escritura alrededor de la periferia de los platos, se arruinaron por las mismas razones que para los discos de cabeza por pista.

Por tanto, se debe aprender a vivir con latencias de búsqueda y rotacional y diseñar en torno a ellas. Poner atención a cómo se organizan los datos en el disco, que se discute al final de la sección 19.2,

Figura 19.4 Reducción del tiempo de búsqueda promedio y la latencia rotacional mediante la realización de accesos a disco fuera de orden.



reduce tanto el tiempo de búsqueda como la latencia rotacional para los patrones de acceso que se encuentran comúnmente. Poner en cola las peticiones de acceso a disco y llevarlas a cabo fuera de orden es otro método útil. El último método es aplicable cuando múltiples programas independientes comparten una memoria de disco (*multiprogramación*) o cuando algunas aplicaciones buscan peticiones de acceso a disco independientes que se pueden regresar fuera de orden (*procesamiento de transacción*). En la figura 19.4 los sectores se piden en el orden A, B, C, D, E, F. Si sucede que la cabeza está cerca del sector C, los sectores se pueden leer en el orden C, F (en la misma pista), D (en la pista vecina), E, B, A, con el orden óptimo calculado al considerar los tiempos de movimiento de cabeza y latencias de rotación con base en las ubicaciones físicas de los diversos sectores.

■ 19.4 Caching de disco

La *caché de disco* constituye una unidad de memoria de acceso aleatorio que se usa para reducir el número de accesos al disco en forma muy parecida a como la caché de CPU reduce los accesos a la memoria principal más lenta. Por ejemplo, cuando se necesita un sector de disco particular, se pueden leer varios sectores adyacentes, o quizá toda una pista, en una caché de disco. Si ocurre que accesos subsiguientes al disco están en sectores adicionales que se leyeron, se pueden satisfacer de la caché mucho más rápida, con lo que se obvia la necesidad de muchos accesos a disco adicionales. Antes del surgimiento de caché o *buffer* hardware dedicados en los controladores de disco, prevalecía una forma de *buffering* de software, donde el sistema operativo realizaba un *caching* similar o función de *lectura anticipada* usando una porción de la memoria principal que se apartaba para contener sectores de disco. Las tres memorias de disco mencionadas en la tabla 19.1 tienen *buffer*/cachés dedicadas que varían en tamaño, desde 0.125 hasta 16 MB.

Puesto que el disco da órdenes de magnitud más lenta que la memoria principal, la operación de una caché de disco se puede controlar completamente por software. Cuando el controlador de disco recibe una petición de acceso a disco, se consulta el directorio del caché de disco para determinar si el sector solicitado está en la caché de disco. Si lo estuviera, el sector se proporciona desde la caché de disco (lee) o modifica ahí (escribe); si no fuera así, se inicia un acceso regular al disco. En virtud de que el tiempo para buscar en el directorio del caché de disco es mucho más lento que el tiempo promedio para un acceso a disco, la cabecera es muy pequeña, mientras que el pago en caso de encontrar el sector en la caché de disco es muy significativo. Las tasas de impacto/fallo para cachés de disco son muy difíciles de encontrar, pues los fabricantes usan diseños propietarios para sus cachés de disco y son reacios a compartir datos de rendimiento detallados. Para propósitos de diseño inicial, una estimación razonable para la tasa de fallo de una caché de disco es 0.1 [Crag96].

Observe que, cuando toda una pista se lee en una caché de disco, la latencia rotacional casi se elimina por completo. Esto último sucede porque, tan pronto como la cabeza se alinea con la pista deseada, la lectura puede comenzar, sin importar cuál sector llega a continuación bajo la cabeza. Los sectores a los que se accede se pueden almacenar en la memoria caché de acceso aleatorio en sus posiciones

correspondientes dentro del espacio asignado a la pista. De igual modo, cuando una pista se debe escribir de vuelta al disco para hacer espacio para una nueva pista, los sectores se pueden copiar en el orden en que los encuentra la cabeza. Observe que el uso de una política de escritura para cachés de disco, que es una elección natural para minimizar el número de accesos a disco, requiere una forma de fuente de energía de respaldo para garantizar que las actualizaciones no se pierden en el evento de falta de energía u otras formas de interrupción.

Ejemplo 19.3: Beneficio al rendimiento del caching de disco Un disco de 40 GB con 100 000 pistas contiene 512 B sectores, rota a 7 200 rpm y tiene un tiempo de búsqueda promedio de 10 ms. Si supone que todas las pistas se llevan a la caché de disco, ¿cuál es el beneficio al rendimiento o aceleración debida al *caching* de disco en una aplicación cuyo tiempo de ejecución está completamente dominado por accesos a disco? En otras palabras, suponga que el tiempo de procesamiento y otros aspectos de la aplicación son despreciables en relación con los tiempos de acceso a datos desde el disco. En sus cálculos, use una tasa de fallo de caché de disco de 0.1.

Solución: Una pista promedio en este disco contiene $40 \times 2^{30} / (100\,000 \times 512) \cong 839$ sectores. Por tanto, un acceso a sector requiere un tiempo promedio de 10 ms más el tiempo necesario para $1/2 + 1/839$ rotaciones. Lo anterior representa 18.35 ms. La lectura de todo un sector implica el tiempo de búsqueda promedio más una rotación completa, o 26.67 ms. El *caching* de disco con una tasa de fallo de 0.1 permite sustituir diez accesos de sector de 18.35 ms con un acceso a pista de 26.67 ms, ello produce una aceleración de $10 \times 18.35 / 26.67 \cong 6.88$. La aceleración real es un poco menor, no sólo porque se ignoraron el tiempo de procesamiento y otros aspectos de la aplicación, también porque el acceso a disco que se satisface desde la caché de disco todavía debe pasar a través de varios intermediarios, como canales, controladores de almacenamiento y controladores de dispositivo, cada uno aporta alguna latencia.

En la discusión anterior se supuso que la caché de disco se incorpora en el controlador de disco. Existen varios inconvenientes a este enfoque:

1. La caché se liga al CPU a través de intermediarios como buses, canal I/O, controlador de almacenamiento y controlador de dispositivo. De este modo, incluso en el caso de un impacto en caché de disco, se incurre en alguna latencia no trivial.
2. En los sistemas con muchos discos, habrá múltiples cachés, con algunas unidades subutilizadas cuando los discos asociados no son muy activos, y otros que no tengan suficiente espacio para mantener todas sus pistas útiles

Un enfoque para resolver estos problemas de lejanía y desequilibrio consiste en usar una caché grande que se coloque más cerca del CPU y comparta su espacio con datos provenientes de muchos discos. Para una capacidad total de caché de disco específica, este último enfoque resulta más eficiente. Se puede incluir tal caché compartida además de las de los controladores de disco, ello conduce a una estructura de *caching* multinivel.

■ 19.5 Arreglos de discos y RAID

A pesar de las mejoras significativas en la capacidad y rendimiento de las unidades de disco duro durante las décadas pasadas, todavía existen aplicaciones que requieren mayor capacidad o ancho de banda de las que incluso las unidades de disco más grandes y rápidas disponibles pueden otorgar. Fabricar discos más grandes y rápidos, aunque técnicamente posible, no es económicamente viable por-

que no se necesitan en cantidades grandes como para generar una economía de escala. Sería preferible encontrar una solución con base en el uso de arreglos de discos baratos que se construyen en grandes volúmenes y económicos, para ofrecer al mercado de computadoras personales capacidad expandida y rendimiento mejorado. A esto se le conoce como enfoque de “*arreglo de disco*”.

Desde luego, es sencillo resolver el problema de capacidad con el uso de un arreglo de discos. Pero superar el límite de rendimiento no es tan sencillo y requiere más esfuerzo. Más aún, las aplicaciones que usan grandes cantidades de datos usualmente también requieren certidumbre en la disponibilidad de datos, integridad y seguridad. Considere un gran banco o sitio de comercio electrónico en la red que use 500 discos para almacenar sus bases de datos y otra información. Cada uno de los 500 discos pueden fallar una vez cada 100 000 horas (11^+ años). Sin embargo, en conjunto, el sistema de 500 discos tiene una tasa de falla que es casi 500 veces mayor (una falla cada ocho días). Por tanto, es imperativo incorporar la tolerancia de fallas por redundancia en cualquier solución de arreglo de discos.

El término RAID, por las siglas en inglés para arreglo redundante de discos o independientes, se acuñó a finales de la década de 1980 para referirse a una clase de soluciones a problemas de capacidad y confiabilidad para aplicaciones que requieren grandes conjuntos de datos [Patt88]. Antes de la formulación del concepto RAID, se había reconocido que los arreglos de disco, con sus muchos mecanismos independientes de lectura/escritura, podían ofrecer ancho de banda de datos mejorada por computadoras vectoriales o paralelas cuyo rendimiento estaba limitado por la tasa a la que los datos se alimentaban a sus unidades de procesamiento enormemente encauzadas o múltiples. A este uso de arreglos de disco para sacar ventaja del ancho de banda agregado de un gran número de discos por los que pasan datos *rayados* (*striping*) se le conoce como “RAID nivel 0”, aunque no se incluyó en la taxonomía original [Patt88] y no tiene redundancia para justificar la designación RAID.

Los arreglos de disco RAID0 se pueden diseñar para operar en modo síncrono o asíncrono. En un arreglo de disco síncrono, la rotación de los múltiples discos y sus movimientos de cabeza están sincronizados de modo que todos los discos acceden al mismo sector lógico local en la misma pista, al mismo tiempo. Externamente, tal arreglo de disco de d unidades aparece como un solo disco que tiene un ancho de banda igual a d veces el de un solo disco. Diseñar tales arreglos de disco requiere modificaciones costosas a unidades de disco del mercado y sólo es efectiva en costo para usar con supercomputadoras de alto rendimiento. En los arreglos de disco asíncronos se accede en forma muy parecida a los bancos de memoria en memoria interpolada. Puesto que múltiples accesos a disco pueden estar en proceso al mismo tiempo, el ancho de banda efectivo aumenta. Algunos sistemas RAID de nivel superior se pueden configurar para operar como unidades RAID0 cuando el usuario no requiere la confiabilidad agregada ofrecida por la redundancia.

El uso de *mirroring*, el almacenamiento de una copia de respaldo de los contenidos de un disco en otro *disco espejo* (*mirror*), permite la tolerancia de cualquier falla de disco individual con un procedimiento de recuperación muy simple: detectar el fallo del disco mediante su codificación de detección de error interconstruida y conmutarlo al correspondiente disco espejo mientras se sustituye el disco dañado. El esquema *mirroring* de RAID1 implica 100% de redundancia, que representa un alto precio por pagar para tolerancia de fallo de disco sencillo cuando se usan decenas o cientos de discos. RAID2 representa un intento por reducir el grado de redundancia a través de códigos de corrección de errores en lugar de duplicación. Por ejemplo, un código Hamming de corrección de error de bit sencillo requiere cuatro bits de verificación para 11 bits de datos. Por tanto, si los datos se rayan a través de 11 discos, cuatro discos serán suficientes para retener la información de verificación (contra 11 en el caso de *mirroring*). Con más discos de datos, la redundancia relativa de RAID2 se reduce aún más.

Es evidente que incluso la redundancia del código Haming es demasiada, y se puede lograr el mismo efecto con apenas dos discos adicionales: un *disco de paridad* (*parity*) que almacena la XOR de los bits de datos de todos los otros discos y un disco *spare* (de repuesto) que pueda tomar el lugar de un disco

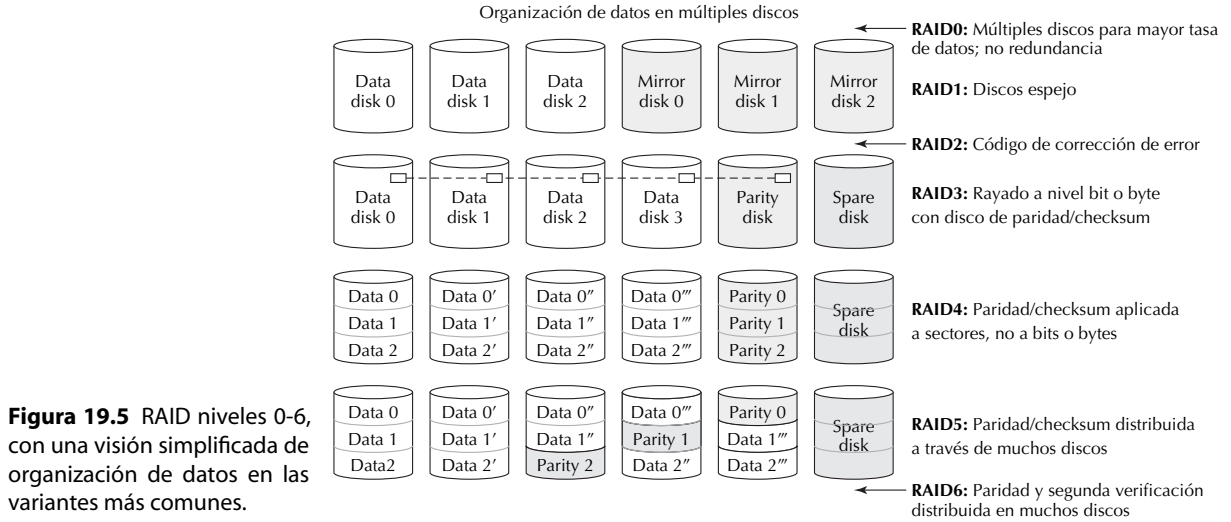


Figura 19.5 RAID niveles 0-6, con una visión simplificada de organización de datos en las variantes más comunes.

dañado cuando se requiera (RAID3). Esto es posible debido a la capacidad de detección de error interconstruida en el nivel sector con base en la verificación de redundancia cíclica incluida rutinariamente. Si hay d discos de datos (*data*) que contengan los valores de bit b_i , $0 \leq i \leq d-1$, en una posición particular en la superficie del disco, el disco de paridad retendrá el siguiente valor de bit en la misma posición:

$$p = b_0 \oplus b_1 \oplus \cdots \oplus b_{d-1} = \bigoplus_{i=0}^{d-1} b_i$$

Hasta la falla del disco de datos j , su correspondiente valor de bit b_j se puede reconstruir a partir de los contenidos de los restantes discos de datos y el disco de paridad, del modo siguiente:

$$b_j = \left(\bigoplus_{i=0, i \neq j}^{d-1} b_i \right) \oplus p$$

Cada uno de los RAID niveles 1-3 (figura 19.5) tiene anomalías que limitan su utilidad o dominio de aplicabilidad.

1. Además del 100% de redundancia, que no puede ser un conflicto mayor con discos baratos, el RAID de nivel 1 involucra una penalización por rendimiento sin importar si se usa el rayado. Cada operación de actualización en un archivo también debe actualizar la copia de respaldo. Si la actualización del respaldo se realiza enseguida, el más grande de los dos tiempos de acceso a disco dictará el retardo. Si las actualizaciones de respaldo se ponen en cola y realizan cuando es conveniente, aumentará la probabilidad de pérdida de datos debido a falla de disco.
2. El RAID de nivel 2 mejora el factor de redundancia del nivel 1 mediante el uso de rayado junto con código Hamming de corrección de error sencillo. Sin embargo, como consecuencia de que se debe acceder a todos los discos para leer o escribir un bloque de datos, se incurre en severas penalizaciones de rendimiento por leer o escribir pequeñas cantidades de datos.
3. El RAID de nivel 3 también tiene el inconveniente de que todos los discos deben participar en cada operación de lectura y escritura, ello reduce el número de accesos a disco que se pueden realizar en tiempo unitario y daña el rendimiento para lecturas y escrituras pequeñas. Con un controlador especial que calcula la paridad requerida para operaciones de escritura y un *buffer* de escritura no volátil para retener los bloques mientras se calcula la paridad, el RAID3 puede lograr rendimiento cercano al de RAID0.

Puesto que a los datos de disco usualmente se accede en unidades de sectores en lugar de bits o bytes, RAID nivel 4 aplica la paridad o *checksum* (suma de verificación) a sectores de disco. En este caso, el rayado no afecta archivos pequeños que encajan por completo en un sector. Las operaciones de lectura se realizan desde discos de datos sin necesidad de ingresar al disco de paridad. Cuando un disco falla, sus datos se reconstruyen a partir de los contenidos de otros discos y el disco de paridad en forma muy similar a RAID3, excepto que la operación XOR se realiza en sectores más que en bits. Una operación de escritura requiere que el sector de paridad asociado también se actualiza con cada actualización de sector e involucra dos accesos al disco de paridad. Con letras mayúsculas para denotar contenidos de sector de datos y paridad, se tiene:

$$P_{nueva} = D_{nueva} \oplus D_{vieja} \oplus P_{vieja}$$

Esta ecuación muestra una ventaja de RAID4 sobre RAID3, pero también revela uno de sus inconvenientes clave: a menos que las operaciones de escritura sean bastante raras, los accesos al disco de paridad crean un severo problema en el rendimiento.

RAID nivel 5 remueve el problema al distribuir los sectores de paridad sobre todos los discos en lugar de colocarlos en un disco dedicado. Como se ve en el ejemplo bosquejado en la figura 19.5, los datos de paridad para los cuatro sectores del archivo 0 se almacenan en el disco de la extrema derecha, mientras que los sectores de paridad para los archivos 1 y 2 se colocan en discos diferentes. Este esquema distribuye los accesos de paridad y resuelven el problema de rendimiento que resulta de actualizaciones de paridad. Las operaciones de lectura son independientes como antes. Por tanto, en RAID5 pueden estar en progreso, al mismo tiempo, múltiples operaciones de lectura y escritura. Por ejemplo, los sectores etiquetados Data0 y Data1' se pueden actualizar al mismo tiempo, pues los sectores y sus paridades asociadas están en discos diferentes.

Observe que los RAID de niveles 3-4, y el más comúnmente usado RAID5, ofrecen protección adecuada contra fallas de disco individual bajo la suposición de que la falla en un segundo disco no ocurre mientras está en progreso la recuperación de una falla. Para proteger contra fallas dobles, los RAID de nivel 6 agregan una segunda forma de verificación de error al esquema *parity/checksum* del nivel 5, al usar el mismo esquema de distribución, pero con los dos sectores de verificación colocados en discos diferentes. En este contexto, se dice que RAID6 usa "redundancia P+Q", donde P representa el esquema de paridad y Q el segundo esquema. Éste puede ser más complejo porque rara vez, si acaso, se invoca para la reconstrucción de datos. Con RAID6 sólo hay peligro de pérdida de datos si ocurre una tercera falla de disco antes de que la primera falla de disco se haya sustituido.

La eficiencia de RAID se puede mejorar al combinar la idea con la de un sistema de archivo estructurado por bitácora (*log-structured*). En un archivo estructurado por bitácora, los datos no se modifican en el lugar; en vez de ello, se crea una copia fresca de cualquier registro modificado en una localidad diferente, y el directorio de archivo se actualiza adecuadamente para reflejar la nueva localidad. Suponga que un controlador RAID tiene una caché de escritura muy grande como para retener el cúmulo de datos de todo un cilindro. Entonces, conforme se modifica un sector, se marca para su borrado de su ubicación actual. La versión actualizada no se escribe de vuelta inmediatamente, sino que se mantiene en la caché no volátil hasta que el cilindro en caché esté lleno. En este punto, el cilindro se escribe a cualquier disco que esté disponible. Dado que los contenidos del cilindro se convierten en repuesto al borrado de sus sectores, se requiere una especie de recolección de basura automática para leer dos o más de tales cilindros durante los tiempos cuando los discos no se usen y para comprimirlos en un nuevo cilindro. Un beneficio adicional de este enfoque es que los datos de cilindro se pueden escribir en formato comprimido, y la descompresión ocurre durante la lectura, ello implica copiar todo el cilindro en la caché de disco.

Los productos RAID comerciales los ofrecen todos los grandes vendedores de almacenamiento masivo, entre ellos IBM, Compaq-HP, Data General, EMC y Storage Tek. La mayor parte de los pro-

ductos RAID ofrecen cierta flexibilidad para seleccionar el nivel RAID con base en las necesidades de aplicación.

19.6 Otros tipos de memoria masiva

Aun cuando la grabación magnética domina la tecnología de elección para memoria secundaria en las computadoras modernas, están disponibles otras opciones. Una característica importante de la memoria de disco, que la convierte en un componente necesario incluso si la memoria principal es bastante grande para todos los programas y datos, es su no volatilidad (vea también la discusión de memoria no volátil en la sección 17.5). Los discos magnéticos pueden retener datos casi indefinidamente, mientras que la memoria principal pierde sus datos cuando la computadora se apaga o la energía se interrumpe por cualquier otra razón. Desde luego, la memoria secundaria no se limita a almacenamiento de datos en línea, de modo que sus contenidos están continuamente disponibles para una computadora; algunas tecnologías de memoria masiva se usan como almacenamiento de respaldo o como medio de grabación para distribución de software, documentos, música, imágenes, películas y otros tipos de contenido.

Antes de proceder con la revisión de los discos ópticos y otras tecnologías de memoria secundaria y terciaria, se debe notar que los *discos flexibles* (*floppy* o *disquetes*) y muchas otras memorias magnéticas rotatorias usan fundamentalmente los mismos principios que los discos duros para la grabación y acceso de datos. En otras palabras, tienen pistas y sectores, actuadores para efectuar movimiento radial de un ensamble de cabeza para alinear con una pista deseada, y cabezas de lectura/escritura que magnetizan pequeñas áreas de la superficie o sienten la dirección de magnetización. Lo que distingue a los discos flexibles de 1.2 MB (figura 3.12*b*), o los discos zip de mayor capacidad (100/250 MB) de los discos duros, es la más baja densidad de grabación y mecanismos de lectura/escritura menos precisa de los primeros, que surgen de la flexibilidad, y tolerancias más amplias, en los medios de grabación y la falta de empaques herméticos que alejen de la superficie de grabación el polvo y otras partículas finas.

Las cintas magnéticas graban datos a lo largo de pistas rectas, en lugar de las pistas circulares de una memoria de disco rotatoria. Una ventaja clave de las cintas magnéticas es que se pueden remover y colocar en archivos permanentes, aunque, en el pasado, muchos tipos de dispositivos de cinta se perdieron por obsolescencia, ello propició que muchos archivos no estuvieran disponibles antes de que las grabaciones mostraran algún tipo de deterioro. Usualmente se graban nueve bits a través de la cinta (un byte de datos, más un bit de paridad; en este contexto, la densidad de bit lineal a lo largo de la cinta es mucho menor que la de los discos duros (por las mismas razones citadas para los discos flexibles). Conforme la cinta se devana de un carrete y se enrolla en el otro, la cabeza de lectura/escritura (y quizá una *cabeza borradora* por separado) vuelan sobre los datos grabados, con técnicas de lectura y escritura muy similares a las de las memorias de disco. Las cintas magnéticas vienen en muchas variedades diferentes: desde pequeños cassettes o cartuchos (figura 3.12*b*) hasta grandes carretes con mecanismo de transporte costosos que pueden mover la cinta a muy alta rapidez y también tienen capacidades de inicio y paro casi instantáneas. Las últimas unidades de cinta usan un *buffer* vacío que contiene un segmento holgado de la cinta para evitar daño al material de ésta y los datos grabados durante el movimiento mecánico a alta rapidez o aceleración/desaceleración.

Conforme los discos flexibles y zip, junto con las cintas magnéticas, gradualmente pierden importancia y se sustituyen por otros medios como los CD para escribir, los discos duros quizá se convertirán en la única memoria de grabación magnética de amplio uso en las computadoras. Hay cierta especulación de que incluso los discos magnéticos duros se desplazarán por memorias ópticas y de semiconductores. Sin embargo, dados los avances pasados y en marcha en la tecnología de memoria de disco, esto no ocurrirá a corto plazo.

Los discos ópticos (CD-ROM, DVD-ROM y sus variedades grabable y regrabable) se han vuelto cada vez más populares desde la década de 1990. En cierta forma, estos dispositivos son similares a las unidades de disco magnético en cuanto a que retienen bloques de datos en la superficie de un plato redondo que debe girar para hacer que los datos estén disponibles a un mecanismo de acceso. Primero se desarrolló el formato de *disco compacto* (CD) y rápidamente se convirtió en el método preferido para distribuir música y productos de software que requerían cientos de megabytes de capacidad. Subsecuentes avances en el material del sustrato, láseres semiconductores, métodos de posicionamiento de cabezas y componentes ópticos condujeron a la creación del formato *disco versátil digital* (DVD) de mayor capacidad, cuya capacidad multigigabyte es suficiente para almacenar una película. Los datos CD y DVD están protegidos a través de sofisticados códigos de corrección de error. Por tanto, la aniquilación de muchos bits grabados (debido a defectos de material o rayones) no afectará el buen uso de los datos recuperados.

Los discos ópticos de sólo lectura (CD, DVD) son descendientes de los primitivos sistemas de almacenamiento óptico que grababan datos mediante la colocación de marcas o perforación de hoyos en tarjetas, cintas de papel y medios similares. Los discos ópticos graban datos mediante la creación de *pits* (depresiones) de tamaño micrométrico en una delgada capa de material óptico (colocada sobre un sustrato plástico) a lo largo de una pista espiral continua que comienza en alguna parte en el interior del disco y termina cerca del borde exterior; para quienes son lo suficientemente viejos para recordar, esto último es lo opuesto de la dirección espiral que se usaba en los discos fonográficos de vinil. Los datos se grababan mediante la creación de *pits* de diversos tamaños durante la producción de un CD o DVD. Un haz de luz láser se enfoca sobre la superficie del plato y las diferencias en la reflexión de la superficie no afectada y los *pits* se detectan con la función de lectura. Observe que lo explicación anterior está muy simplificada y no explica el funcionamiento del diodo láser, el divisor del haz, las lentes y el detector de la figura 19.6 [Clem00].

Los CD y DVD escribibles pueden ser de una sola escritura o reescribibles. En las versiones de una sola escritura, los *pits* se crean al hacer estallar un recubrimiento especial colocado sobre una superficie reflectora (éste es diferente del recubrimiento protector de la figura 19.6), fundir los hoyos en una delgada capa de material especial con el uso de un láser de escritura más potente o por una diversidad de otros métodos. Las versiones reescribibles se apoyan en la modificación reversible de las propiedades ópticas o magnéticas del material mediante la exposición selectiva a luz láser [Clem00].

Los dispositivos *microelectromecánicos* (MEM) se desarrollaron para ofrecer memoria masiva en chips IC. Una tecnología, que sigue IBM, opera como un fonógrafo y usa afiladas puntas de delgadas

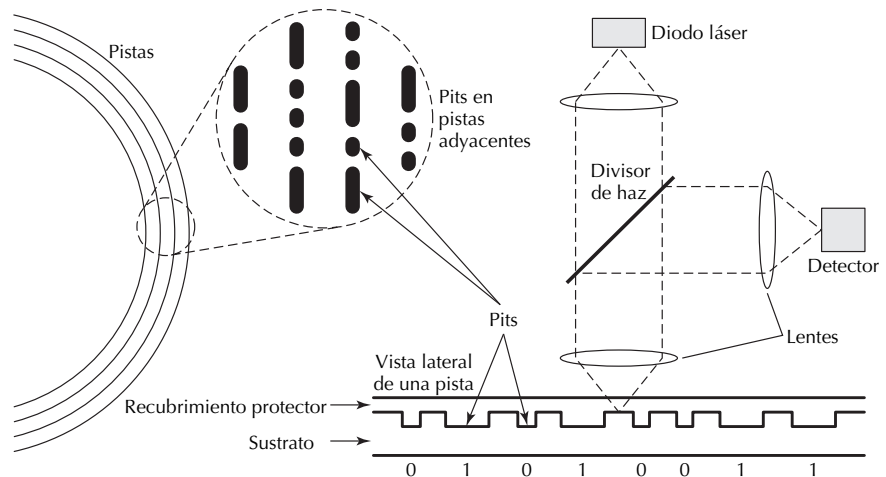


Figura 19.6 Vista simplificada del formato de grabación y mecanismo de acceso para datos en un CD-ROM o DVD-ROM.

ménsulas de silicio para inscribir datos en, y leer desde, un medio polímero [Vett02]. El desarrollo de esa tecnología acercará más a la construcción de verdaderos sistemas de cómputo de un solo chip.

Como ya se mencionó, las unidades de memoria masiva *en línea*, como los discos duros, son accesibles directamente a la computadora, mientras que los medios de almacenamiento *fuera de línea*, como los CD y cintas, se deben colocar o montar en un dispositivo en línea para hacerlos accesibles. Lo intermedio entre ambos son las memorias masivas *cerca de línea* que ofrecen gran cantidad de almacenamiento sin necesidad de intervención humana en el montaje del medio. Con frecuencia tales dispositivos se usan como almacenamiento terciario para retener bases de datos rara vez usadas y otra información que, no obstante, debe estar disponible cuando surja la necesidad. Por ejemplo, las *bibliotecas de cinta automatizadas* ofrecen muchos terabytes de capacidad de almacenamiento con tiempo inferior a un minuto. Usualmente miles de cartuchos de cinta se almacenan en un gran gabinete, con brazos robóticos capaces de elegir un cartucho, cargarlo en un ensamble de lectura/escritura y más tarde colocarlo de vuelta en su rendija de origen.

Sin importar qué tipo de almacenamiento masivo se use, persiste el problema de la permanencia de los datos digitales. Las cintas magnéticas duran pocas décadas. Los CD son mejores, pues su tiempo de vida se mide en centurias. Desde luego, un reto diferente es toparse con dispositivos que lean CD de siglos de antigüedad, dado el rápido ritmo de progreso tecnológico. Los lenguajes de programación y los formatos de las bases de datos también enfrentan el problema de la extinción, ello hace más difícil preservar datos [Tris02].

PROBLEMAS

19.1 Maximización de capacidad de disco

Considere el siguiente problema de optimización en el diseño de un sistema de memoria de disco. El diámetro D de la pista exterior es una constante dada. Usted debe elegir el diámetro xD de la pista más interna, donde $x < 1$. La negociación es que la capacidad de todas las pistas debe ser la misma, de modo que mientras más corta sea la pista interior, menos datos se pueden grabar en cada pista, pero habrá más pistas. ¿Cómo puede derivar el valor óptimo de x y cuáles parámetros de disco o grabación de datos influyen este valor óptimo? Establezca todas sus suposiciones claramente.

19.2 Parámetros de unidades de disco comerciales

Para cada uno de los tres discos mencionados en la tabla 19.1:

- Derive el número promedio aproximado de pistas de disco completo que se pueden contener en su caché. Ignore la cabecera resultante del almacenamiento de la información de identificación, como los números de pista o sector, en el *buffer*.
- Calcule el tiempo necesario para leer un archivo de 10 MB que se almacena en pistas adyacentes de acuerdo con el esquema de la figura 19.3, de modo

que no haya desperdicio de tiempo debido a latencia rotacional entre la lectura de sectores lógicos sucesivos del archivo.

19.3 Tendencias en el costo de discos

En www.cs.utexas.edu/users/dahlin/techTrends/data/diskPrices están disponibles precios de datos para memorias de disco.

- Grafique las variaciones de precio para los discos de computadora personal con el rango de tiempo 1980-2010 en el eje horizontal y el rango de precio \$0-\$3000 en el eje vertical. Grafique una curva separada para cada una de las capacidades 10 MB, 20 MB, 80 MB, 210 MB, 420 MB, 1.05 GB, 2.1 GB, 4.2 GB, 9.1 GB, 18.2 GB, 36.4 GB y 72.8 GB, donde una curva se extienda en el tiempo desde que estuvo disponible por primera vez en el comercio hasta que se discontinuó el producto. Donde se citen múltiples precios para una capacidad de disco dada, elija el más bajo.
- Elabore una gráfica de dispersión con base en los datos de la parte a), donde el eje horizontal cubra de nuevo el marco temporal 1980-2010 y el eje vertical muestre el precio por gigabyte de capacidad, que varía de \$1 a \$1 M en una escala logarítmica (cada

línea de la retícula es un factor diez veces mayor que la anterior).

- c) Discuta las tendencias generales observadas en las partes a) y b) e intente extrapolar en lo futuro más allá de los datos disponibles.

19.4 Distancia de búsqueda promedio

La distancia de búsqueda promedio constituye el número promedio de cilindros a cruzar para llegar al siguiente cilindro de interés. Considere un disco de c cilindros, que tiene una distancia de búsqueda máxima de $c - 1$.

- Si la cabeza retrae al cilindro hogar 0 (el cilindro más exterior) después de cada acceso, entonces ¿cuál es la distancia de búsqueda promedio por acceso?
- Si pudiese elegir un cilindro hogar distinto al cilindro 0, ¿cuál elegiría y por qué?
- Demuestre que, si a la cabeza se le permite permanecer en el último cilindro al que se accede, en lugar de retraerse a un cilindro hogar, entonces la distancia de búsqueda promedio con accesos completamente aleatorios es aproximadamente $c/3$.

19.5 Fórmula de tiempo de búsqueda de disco

Derive una forma simplificada de la fórmula del tiempo de búsqueda de disco (sección 19.3) para la cual $\beta = 0$, y suponga que la cabeza se acelera a una tasa constante para la primera mitad de la distancia de búsqueda y luego desacelera a la misma tasa para la segunda mitad. *Sugerencia:* A una aceleración constante a , la distancia recorrida se relaciona con el tiempo t mediante $d = \frac{1}{2}at^2$.

19.6 Tiempo de búsqueda de disco

Para cada una de las memorias de disco citadas en la tabla 19.1, determine los coeficientes de la fórmula del tiempo de búsqueda (sección 19.3), si se supone que el tiempo de búsqueda promedio citada será para la mitad de la máxima distancia de cilindro.

19.7 Cronograma de acceso a disco

Un disco con una sola superficie de grabación y 10000 pistas tiene un controlador que optimiza el rendimiento a través de procesamiento extraordinario de las peticiones de acceso. La cola de peticiones de acceso actualmente retiene los siguientes números de pista (en orden de llegada): 173, 2939, 1827, 3550, 1895, 3019, 2043, 3502, 259, 260. La cabeza acaba de completar la lectura

de la pista 250 y está en camino hacia la pista 285 para un acceso de lectura. Determine la distancia de viaje total de la cabeza (en pistas) para cada una de las siguientes alternativas de cronograma.

- First come, first served* (primero en llegar, primero en atender) (FCFS).
- Tiempo de búsqueda más corto primero (SSTF = *shortest seek time first*).
- Tiempo de búsqueda más corto entre las tres peticiones más antiguas.
- Mantener dirección de digitalización (si, por decir, la cabeza se mueve hacia adentro, el movimiento continúa en la misma dirección hasta que se accede a la pista solicitada más interna; luego invierte la dirección del movimiento de cabeza).

19.8 Caching de disco y cronograma de acceso

En muchas aplicaciones, a las pistas en un disco no se accede uniformemente; en vez de ello, un pequeño número de pistas usadas con frecuencia representan una gran fracción de los accesos a disco. Suponga que se sabe que 50% de todos los accesos a disco se realizan a 5% de las pistas.

- ¿Esta información se puede usar para hacer más efectivo el esquema de *caching* de disco?
- ¿Cuál de los algoritmos de cronograma de disco dados en el problema 19.7 quizá se desempeñará mejor en la optimización de los accesos a disco y por qué?
- ¿Puede imaginar un algoritmo de cronograma diferente que se desempeñe mejor que el de la parte b)?

19.9 Parámetros de rendimiento de disco

Un disco de un solo plato gira a 3600 rpm y tiene 256 cilindros, cada uno almacena 256 MB. El tiempo de búsqueda está dado por la fórmula $s + tc$, donde $s = 0.5$ ms es el tiempo de arranque, $t = 0.05$ ms es el tiempo de viaje de la cabeza de un cilindro a uno adyacente y c es la distancia del cilindro; por ejemplo, si la cabeza debe viajar cinco cilindros hacia adentro, entonces se necesitan 0.75 ms para llegar ahí.

- Si se ignoran las brechas entre cilindros y otras cabezas, calcule la tasa de transferencia de datos pico.
- Si supone una distancia de cilindro promedio de 85 desde un acceso a un cilindro elegido aleatoriamente, calcule el tiempo de acceso promedio para un sector de 4 KB.

- c) Con base en su respuesta a la parte *b*), ¿cuánto más rápido en promedio es acceder a un sector aleatorio en la misma pista que a otro en cualquier parte del disco?
- d) ¿Cuál es el tiempo mínimo necesario para respaldar todos los contenidos del disco en un almacenamiento terciario?

19.10 Tecnología de memoria de disco

Se puede vislumbrar una solución intermedia entre discos con una sola cabeza en movimiento por plato y discos de cabeza por pista, que tiene una cabeza dedicada para cada pista para eliminar por completo el tiempo de búsqueda. Imagine un disco que tiene ocho cabezas montadas en un ensamblado. Éstas se mueven juntas y se alinean con diferentes pistas de modo que la tasa de datos durante lectura y escritura puede representar ocho veces la de un disco de una sola cabeza.

- a) ¿Cuál sería el mejor ordenamiento de las cabezas respecto de las pistas? En otras palabras, ¿las cabezas deben leer/escribir pistas adyacentes o pistas que estén separadas?
- b) Enumere los beneficios de rendimiento, para el caso de que hubiera alguno, de su diseño de disco de la parte *a*).
- c) Especule por qué tales discos no se usan en las PC ordinarias.

19.11 Sistemas RAID

Suponga que cada uno de los discos mencionados en la tabla 19.1 se usarán para construir un sistema RAID de nivel 5 que encaje en un gabinete más grande que tenga un volumen de 1 m^3 . Casi la mitad del espacio del gabinete ocupará brechas de ventilación, ventiladores y equipo de suministro de energía. Para cada tipo de disco:

- a) Estime la capacidad de almacenamiento total máxima que se puede lograr.
- b) Derive el rendimiento total de lectura máximo del sistema RAID; establezca todas sus suposiciones.
- c) Repita la parte *b*) para el rendimiento total de escritura.

19.12 Sistemas RAID

Considere las organizaciones de datos RAID4 y RAID5 que se muestran en la figura 19.5

- a) Si cada acceso al disco para lectura de datos involucra un solo sector, ¿cuál organización conduce a un ancho de banda más alto para lectura? Cuantifique la diferencia para distribución aleatoria de direcciones de lectura.
- b) Repita la parte *a*) para accesos de escritura.
- c) Suponga que uno de los discos falla y sus datos se deben reconstruir en el disco de repuesto. ¿Cuál organización ofrece mejor rendimiento para lecturas durante el proceso de reconstrucción?
- d) Repita la parte *c*) para accesos de escritura.

19.13 Parámetros y rendimiento de disco flexible

Considere un dispositivo de disco flexible de 3.5 pulgadas que use discos de dos lados con 80 pistas por lado. Cada pista contiene nueve sectores de 512 B. El disco gira a 360 rpm y el tiempo de búsqueda constituye $s + tc$, donde $s = 200 \text{ ms}$ representa la cabecera *start/stop* y $t = 10 \text{ ms}$ significa el tiempo de viaje entre pista y pista.

- a) Calcule la capacidad del disco flexible.
- b) Determine el tiempo de acceso promedio a un sector aleatorio si la posición de origen de la cabeza está en la pista más externa.
- c) ¿Cuál es la tasa de transferencia de datos cuando el sector se haya posicionado?

19.14 Parámetros de disco flexible

Considere un disco flexible, anote su capacidad y vea si es de uno o dos lados. Luego desmantélelo para medir los parámetros (radio interior, radio exterior) de su área de grabación. Entonces, si supone que el floppy tiene 80 pistas por lado de igual capacidad, determine sus parámetros, como densidad de pista, de grabación lineal y de promedio de grabación por área.

19.15 Parámetros de cinta magnética

Los datos se escriben en cinta magnética usando bloques grabados contiguamente con una densidad de grabación lineal de 1000 b/cm y una brecha de 1.2 cm entre bloques. Recuerde que una cinta usualmente tiene ocho pistas de grabación de datos y una de paridad.

- a) ¿Qué fracción de la capacidad bruta total de la cinta retiene datos útiles si cada bloque contiene un registro de 512 B?
- b) Repita la parte *a*), pero esta vez suponga que cuatro registros están empacados en un bloque de cinta.

19.16 Parámetros CD

Un CD tiene un área de grabación con radio exterior (interior) aproximado de 6 cm (2 cm). Retiene 600 MB de datos en una pista espiral con densidad de bit fija y separación de pista de 1.6 μm . Ignore los efectos de operación preparatoria (*housekeeping*) y cabeceras de verificación de error. Derive la densidad de bit lineal y densidad de grabación por área para el CD.

19.17 Parámetros DVD

Un DVD tiene un área de grabación con radio exterior (interior) aproximado de 6 cm (2 cm). Retiene 3.6 GB de datos en una pista espiral con densidad de bit fijo y separación de pista de 1.6 μm . Ignore los efectos de *housekeeping* y cabeceras de verificación de error.

- Derive la densidad de bit lineal y densidad de grabación por área para tal DVD.
- Si supone que no hay compresión de datos, ¿de qué tamaño es el video que se puede almacenar en este DVD si cada imagen tiene 500×800 pixeles, cada pixel requiere 2 B de almacenamiento y la imagen se debe regenerar 30 veces por segundo?
- Con base en el resultado de la parte b), ¿qué puede decir acerca de la razón de compresión promedio para los formatos de las películas DVD comunes?

- ¿A qué rapidez debe girar el DVD si se debe lograr la tasa de datos requerida para la parte c)?

19.18 Memoria terciaria estilo rocola

Considere una rocola que almacena hasta 1024 cartuchos ópticos y tiene un mecanismo para elegir y “tocar” cualquier cartucho. El tiempo de selección y establecimiento es de 15 s. La reproducción a alta rapidez para leer los datos en el cartucho tarda 120 s. El cartucho tiene un área de grabación útil de 100 cm^2 .

- Derive la densidad de grabación por área requerida si este sistema debe ofrecer 1 TB de capacidad.
- ¿Qué tasa de transferencia de datos se implica por la información dada para el sistema de la parte a)?
- ¿Cuál es la tasa de datos promedio efectiva cuando muchos cartuchos se leen en sucesión?
- Sugiera una aplicación para esta memoria terciaria estilo rocola y discuta cuáles otros componentes necesitaría para hacerla funcionar para su aplicación sugerida.
- Especule acerca de la posibilidad de un sistema de almacenamiento petabyte con la misma operación estilo rocola.

REFERENCIAS Y LECTURAS SUGERIDAS

- | | |
|----------|--|
| [Asha97] | Ashar, K., <i>Magnetic Disk Drive Technology</i> , IEEE Press, 1997. |
| [Clem00] | Clements, A., <i>The Principles of Computer Hardware</i> , Oxford, 3a. ed., 2000. |
| [Crag96] | Cragon, H. G., <i>Memory Systems and Pipelined Processors</i> , Jones and Bartlett, 1996. |
| [Dani99] | Daniel, E. D., C. D. Mee y M. H. Clark, eds., <i>Magnetic Recording: The First 100 Years</i> , IEEE Press, 1999. |
| [Gang94] | Ganger, G. R., B. R. Worthington, R. Y. Hou y Y. N. Patt, “Disk Arrays: High-Performance, High-Reliability Storage Subsystems”, <i>IEEE Computer</i> , vol. 27, núm. 3, pp. 30-36, marzo de 1994. |
| [Henn03] | Hennessy, J. L., y D. A. Patterson, <i>Computer Architecture: A Quantitative Approach</i> , Morgan Kaufmann, 3rd ed., 2003. |
| [Patt88] | Patterson, D. A., G. A. Gibson y R. H. Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID)”, <i>Proceedings of the ACM SIGMOD Conference</i> , Chicago, junio de 1988, pp. 109-116. |
| [Sier90] | Sierra, H. M., <i>An Introduction to Direct Access Storage Devices</i> , Academic Press, 1990. |
| [Tris02] | Tristram, C., “Data Extinction”, <i>Technology Review</i> , vol. 105, No. 8, pp. 36-42, 2002. |
| [Vett02] | Vettiger, P., et al., “The ‘Millipede’—Nanotechnology Entering Data Storage”, <i>IEEE Trans. Nanotechnology</i> , vol. 1, núm. 1, pp. 39-55, Vea también el sitio web de IBM, en: www.research.ibm.com/pics/nanotech |

■ CAPÍTULO 20

MEMORIA VIRTUAL Y PAGINACIÓN

“Idealmente, sería deseable una capacidad de memoria muy grande, de manera que cualquier palabra particular estará inmediatamente disponible... Estamos... forzados a reconocer la posibilidad de construir una jerarquía de memorias, cada una tendrá mayor capacidad que la anterior, pero al menos será accesible rápidamente.”

Burkes, H. Goldtine y J. von Neumann, en un informe ahora clásico, 1946

“Virtualmente en cualquier lado, este viernes.”

De los anuncios publicitarios previos al estreno de la película S1m0ne, que presenta a una estrella cinematográfica virtual, 2002

TEMAS DEL CAPÍTULO

- 20.1** Necesidad de la memoria virtual
- 20.2** Traducción de dirección en memoria virtual
- 20.3** *Translation lookaside buffer*
- 20.4** Políticas de sustitución de página
- 20.5** Memorias principal y masiva
- 20.6** Mejora del rendimiento de la memoria virtual

Cuando la memoria principal no tiene la capacidad para retener un programa muy grande junto con sus datos asociados, o muchos programas más pequeños que necesiten coexistir en la máquina, se fuerza a romper los objetos en pedazos y moverlos dentro y fuera de la memoria principal de acuerdo con sus patrones de acceso. En una memoria virtual, el proceso de movimiento de datos dentro y fuera de la memoria principal está automatizado, de modo que el programador o compilador no necesitan preocuparse por ello. Lo que el programador/compilador observa es un gran espacio de dirección, aun cuando, en algún momento específico, sólo una pequeña parte de este espacio esté mapeada a memoria principal. En este capítulo se revisan estrategias para el movimiento de datos entre memorias secundaria y primaria, se observa el problema de la traducción de dirección virtual a real y las formas de acelerarla, además se discute el rendimiento de la memoria virtual.

■ 20.1 Necesidad de la memoria virtual

Mientras que las caché se usan para mejorar el rendimiento de una forma transparente, la memoria virtual se usa sólo por conveniencia. En algunos casos, deshabilitar la memoria virtual (si se permite)

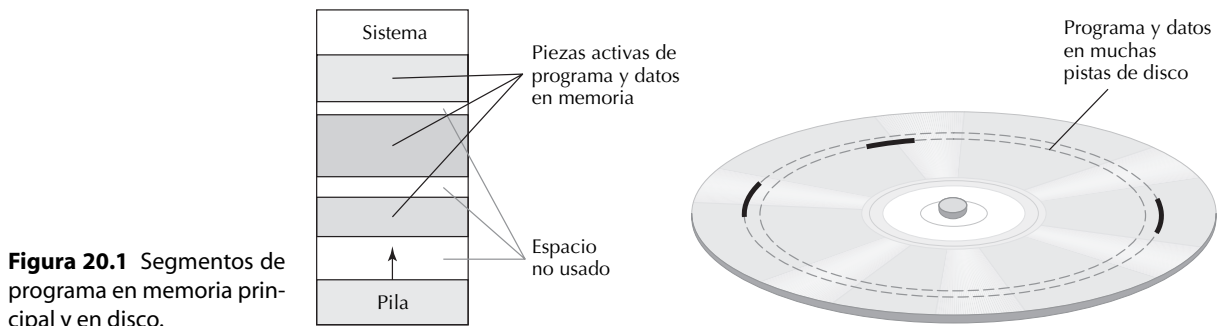


Figura 20.1 Segmentos de programa en memoria principal y en disco.

puede conducir a mejor rendimiento. La memoria virtual da la ilusión de una memoria principal que es mucho mayor que la memoria física disponible. Los programas que se escriben usando un gran espacio de dirección virtual se pueden ejecutar en sistemas con cantidades diversas de memoria real (física). Por ejemplo, MiniMIPS usa direcciones de 32 bits, pero el sistema casi nunca tiene 4 GB de memoria principal. Aun así, el hardware de gestión de memoria virtual da la ilusión de un espacio de dirección física de 4 GB con, o sea, una unidad de 64 MB de memoria principal.

En ausencia de memoria virtual, un programador o software de sistema debe gestionar explícitamente el espacio de memoria disponible, y mover, según requiera, los pedazos de un programa grande y su conjunto de datos dentro y fuera de la memoria principal. Éste es un proceso complicado y proclive al error, además crea problemas de asignación de almacenamiento no triviales porque las unidades de programa y de datos no tienen el mismo tamaño. Con la memoria virtual, el programador o compilador genera código con la intención de que la memoria principal sea tan grande como sugiere el espacio de dirección disponible. Conforme las piezas del programa o sus datos de referencia, el sistema de memoria virtual junta dichas piezas desde la memoria secundaria (disco) hacia la más pequeña memoria principal y, quizá, deja fuera otras piezas. La figura 20.1 muestra un gran programa que ocupa muchas pistas en un disco, con tres piezas (designadas con arcos gruesos en la superficie del disco) residentes en la memoria principal. Como fue el caso para los cachés de procesador, la transferencia de datos entre memorias principal y secundaria es completamente automática y transparente.

En la memoria virtual, la transferencia de datos entre las memorias de disco y principal se realiza mediante unidades de tamaño fijo conocidas como *páginas*. Una página usualmente está en el rango de 4-64 KB. En un disco con 512 B sectores, una página corresponde a los sectores 8-128. La memoria virtual se puede implementar, y se ha hecho, mediante unidades de datos de tamaño variable conocidas como *segmentos*. Sin embargo, como consecuencia de que la implementación de memoria virtual a través de *paginación* es tanto más simple como más prevalente que la *segmentación*, la discusión se limita a la primera.

La figura 20.2 presenta juntas las ideas de memoria virtual y memoria caché, y bosqueja los movimientos de datos entre varios niveles de la jerarquía de memoria y los nombres asociados con las unidades de transferencias de datos. A partir de la figura 20.2, resulta evidente la razón detrás de la elección de los términos “línea” y “página”.

La memoria virtual es una buena idea virtualmente (sin intención de juego de palabras) por las mismas razones que hicieron exitosas a las cachés; esto es, la localidad espacial y temporal de las referencias de memoria (a este punto se regresará en la sección 20.4). Debido a las unidades mucho más grandes (páginas) que se llevan a la memoria principal y el mayor tamaño de la memoria principal, la tasa de impacto en la memoria principal (99.9% o más) usualmente es mucho mejor que la tasa de impacto en cachés (usualmente 90-98%). Cuando una página no se encuentra en la memoria principal,

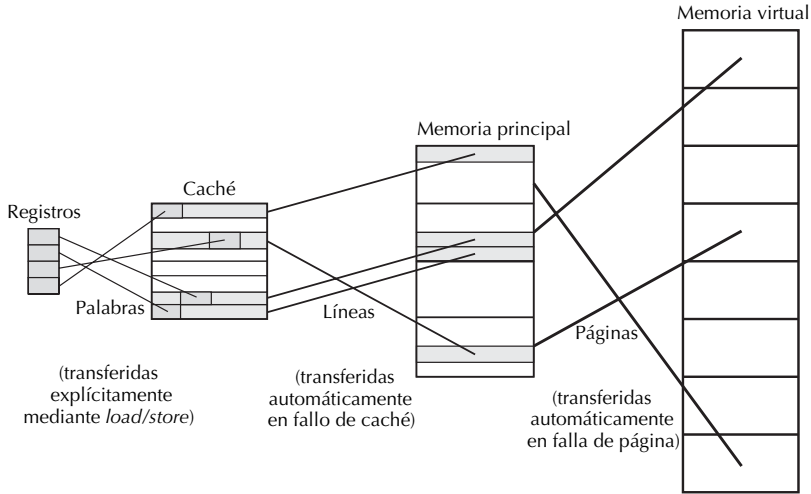


Figura 20.2 Movimiento de datos en una jerarquía de memoria.

se dice que ocurrió una *falla de página*. El uso de “falla de página” se realiza por razones históricas; habría sido más apropiado “fallo de memoria principal” (análogo a “fallo de caché”).

¿Cómo se relaciona la memoria virtual con la memoria caché, que se estudió en el capítulo 18? La caché se usó para hacer que la memoria principal pareciera más rápida. La memoria virtual se usa para propiciar que la memoria principal parezca más grande. Por tanto, la combinación jerárquica de memorias caché, principal y secundaria, con transferencia automática de datos entre ellas, crea la ilusión de una memoria principal muy rápida (comparable en rapidez a SRAM) cuyo costo por gigabyte es un pequeño múltiplo del costo correspondiente para un disco.

20.2 Traducción de dirección en memoria virtual

Bajo memoria virtual, un programa en ejecución genera direcciones virtuales para accesos de memoria. Estas direcciones virtuales se deben *traducir* a direcciones de memoria físicas antes de que las instrucciones o datos se recuperen y envíen al procesador. Si las páginas contienen 2^P bytes y la memoria es direccionable por byte; en consecuencia, una dirección virtual de V bits se puede dividir en un *offset* de byte de P bits dentro de la página y un número de página virtual de $V - P$ bits. En virtud de que las páginas se llevan a la memoria principal en su totalidad, el *offset* de byte de P bits nunca cambia. Por tanto, es suficiente traducir el número de página virtual a un número de página físico correspondiente a la localidad de la página en la memoria principal. La figura 20.3 bosqueja los parámetros de *traducción de dirección virtual a física*.

Ejemplo 20.1: Parámetros de memoria virtual Un sistema de memoria virtual con direcciones virtuales de 32 bits usa 4 KB páginas y una memoria principal direccionable por byte de 128 MB. ¿Cuáles son los parámetros de dirección en la figura 20.3?

Solución: Las direcciones de memoria física para una unidad de 128 MB tienen 27 bits de ancho ($\log_2 128M = 27$) y el *offset* de byte en página tiene 12 bits de ancho ($\log_2 4K = 12$). Esto último deja $32 - 12 = 20$ bits para el número de página virtual y $27 - 12 = 15$ bits para el número de página físico. La memoria principal de 128 MB tiene espacio para $2^{15} = 32K$ páginas, mientras que el espacio de dirección virtual contiene $2^{20} = 1M$ páginas.

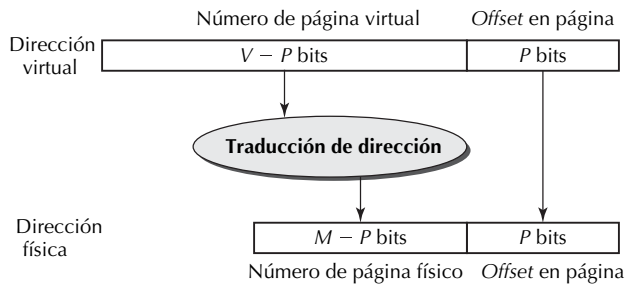


Figura 20.3 Parámetros de traducción de dirección virtual a física.

El esquema de mapeo que se usa para colocar páginas en memoria principal es *completamente asociativo*, ello significa que cada página se puede mapear hacia cualquier lugar en la memoria principal. Lo anterior elimina los *fallos de conflicto* y minimiza las fallas de página, que son extremadamente costosas en términos de penalización de tiempo (unos cuantos milisegundos pueden abarcar millones de ciclos de reloj). Sin embargo, existen *fallos obligatorios* y *fallos de capacidad*, como se discutió para cachés al final de la sección 18.2.

Puesto que la comparación paralela de la etiqueta (tag) de página con muchos miles de tags es muy compleja, con frecuencia se usa un proceso de dos etapas para acceso a memoria: se consulta una tabla de página para encontrar si una página requerida está en memoria y, de ser así, dónde se ubica; entonces se realiza el acceso a memoria real o, en el evento de una falla de página, se inicia el acceso desde disco. En virtud de que los accesos de disco son muy lentos, el sistema operativo salva todo el contexto del programa y cambia a una tarea diferente (*cambio de contexto*) en lugar de forzar al procesador a esperar en forma inactiva durante millones de ciclos hasta que el acceso a disco se complete y la página requerida se lleve a la memoria principal. No obstante, para el caso de fallos de caché, el procesador se atasca durante varios ciclos de reloj hasta que los datos se llevan al caché.

La figura 20.4 muestra una tabla de página y cómo se usa en el proceso de traducción de dirección. Mientras un programa está en ejecución, la dirección de inicio de su tabla de página se almacena en un *registro de tabla de página* especial. El número de página virtual se usa como índice en la tabla de página y se lee la entrada correspondiente. La dirección de memoria para la entrada de tabla de página deseada se obtiene al sumar el número de página virtual (*offset*) a la dirección base contenida en el registro de tabla de página. Como se muestra en la figura 20.4, cada entrada de tabla de página tiene un bit válido, muchas otras banderas (*flags*) y un puntador a los “paraderos” de la página.

La entrada de tabla de página así obtenida puede ser de uno de dos tipos. Si el bit válido se fija, la parte de dirección de la entrada representa un número de página físico que identifica la localidad de la página en la memoria principal (impacto de página). Si el bit válido se restablece, la parte de dirección apunta hacia la localidad de la página virtual en la memoria de disco (fallo [*miss*] de página o falla [*fault*] de página). Además de las páginas válidas e inválidas, que se distinguen mediante el bit válido, una página dentro del espacio de dirección de la memoria virtual puede estar *desadjudicada* (*unallocated*). Esto último sucede porque el espacio de dirección virtual usualmente es grande y la mayoría de los programas no usan todo su espacio de dirección. Una página desadjudicada está completamente vacía de información; incluso no ocupa espacio en el disco.

Cada entrada de tabla de página representa una palabra (4 bytes), que es adecuado para especificar un número de página física o una dirección de sector en disco. Cada vez que una página se copia en la memoria principal o se envía al disco para hacer espacio para otra página, su entrada de tabla de página asociada se actualiza para reflejar la nueva localidad de la página.

Además del “bit válido”, la entrada de tabla de página puede tener un “bit sucio” (para indicar que la página se modificó desde que se llevó a la memoria principal y se debe inscribir de vuelta al disco si fuera necesario sustituirla) y un “bit de uso”, que se fija en 1 siempre que se accede a la página.

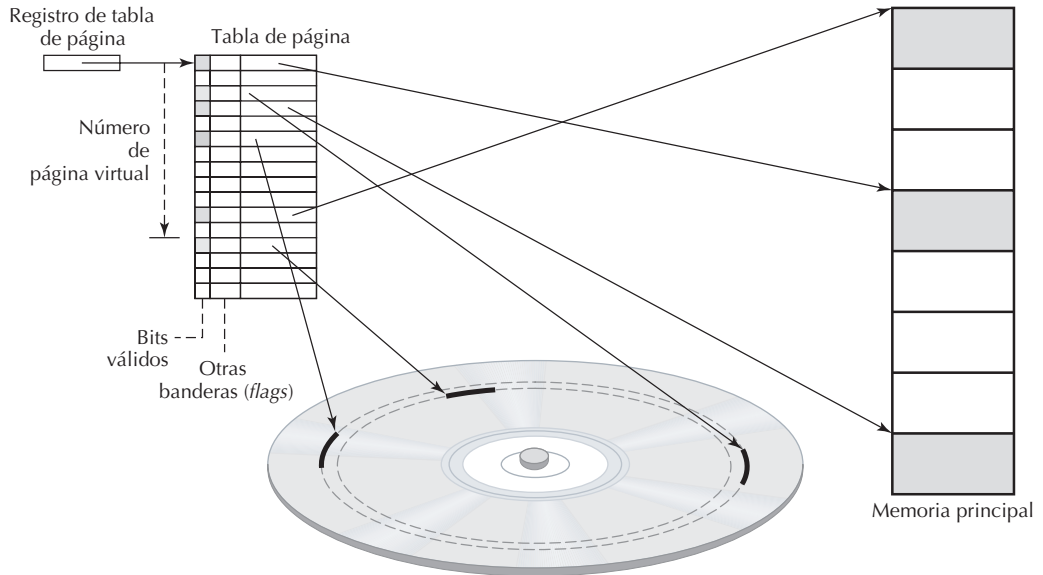


Figura 20.4 El papel de la tabla de página en el proceso de traducción de dirección virtual a física.

El bit de uso encuentra aplicación en la implementación de una versión aproximada de la política de sustitución de usado menos recientemente (LRU) (sección 20.4).

A partir de la discusión previa, resulta evidente que la conveniencia de la memoria virtual viene con cierta cabecera, que consiste de espacio de almacenamiento para tablas de página y un acceso a tabla de página (traducción de dirección) para cada referencia de memoria. En la sección 20.3 se verá que, en la mayoría de los casos, es posible evitar la referencia de memoria adicional; por tanto, hace que la cabecera de tiempo sea despreciable. La cabecera de almacenamiento para tablas de página también se puede reducir sustancialmente. Observe que la tabla de página puede ser grande: en un espacio de dirección virtual de 2^{32} B con páginas de 4 KB, habrá 2^{20} páginas y el mismo número de entradas a tabla de página. En lugar de asignar 4 MB de memoria a la tabla de página, se le puede organizar como una estructura de tamaño variable que crece a su máximo tamaño sólo si es necesario. Por ejemplo, en lugar de una tabla de página con direccionamiento directo, donde la dirección de entrada se determina mediante la suma de la dirección base de la tabla de página al número de página virtual, se puede usar una tabla de ruidos (*hash*) que necesita mucho menos entradas para la mayoría de los programas.

Además de simplificar la gestión de memoria (incluida asignación, vinculación y carga), la memoria virtual ofrece ventajas que pueden hacer que su uso sea útil, aun cuando no se necesite un espacio de dirección de memoria expandida. Dos de los más importantes beneficios adicionales son la *compartición* y la *protección de memoria* (figura 20.5):

1. La compartición de una página por dos o más procesos es directa, sólo requiere que las entradas en las tablas de página asociadas tengan la misma dirección de la página en la memoria principal o en disco. La página compartida puede, y con frecuencia lo hace, ocupar diferentes partes en los espacios de dirección virtual de los procesos de compartición.
2. La entrada de la tabla de página se puede aumentar con *bits de permiso* que especifiquen qué tipos de acceso a la página se permiten y qué otros privilegios se garantizan. En virtud de que todos los accesos a memoria pasan a través de la tabla de página, las páginas que están fuera de los límites a un proceso automáticamente se vuelven inaccesibles a él.

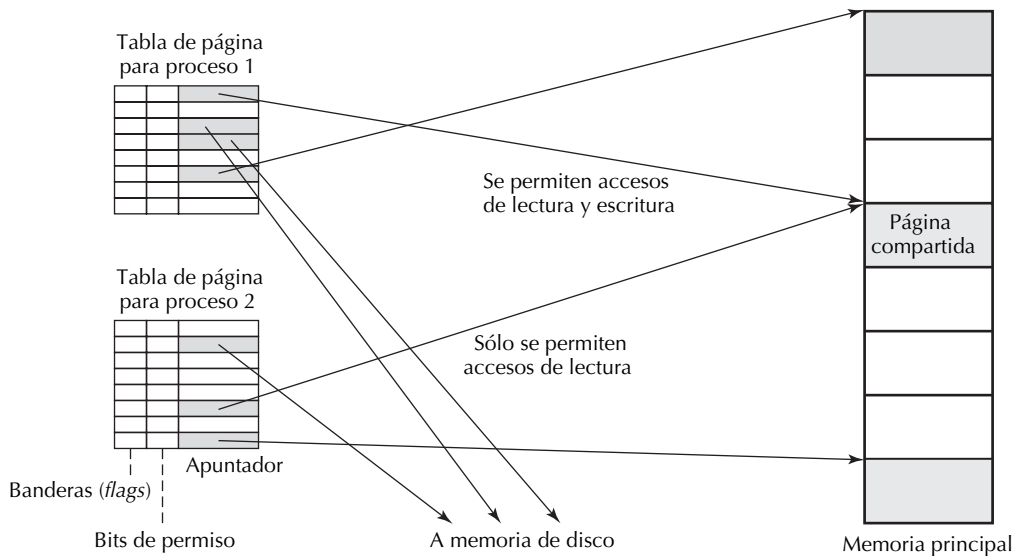


Figura 20.5 Memoria virtual como un facilitador de compartición y protección de memoria.

Aun cuando el espacio de dirección virtual usualmente sea mucho más grande que el espacio de dirección físico (memoria principal), con base en los beneficios apenas discutidos, tiene sentido tener una dirección virtual que sea más estrecha que, o del mismo ancho que, la dirección física.

20.3 Translation lookaside buffer

El acceso a tabla de página para la traducción de dirección virtual a física en esencia duplica al retardo de acceso a la memoria principal. Esto último sucede porque leer o escribir una palabra de memoria requiere dos accesos: uno a la tabla de página y otro a la palabra misma. Con el propósito de evitar esta penalización de tiempo, que reduce la eficiencia de la memoria virtual, se toma ventaja de las propiedades de localidades espacial y temporal de los programas. Puesto que las direcciones consecutivas generadas por un programa con frecuencia residen en la misma página, es probable que el proceso de traducción de dirección consulte la misma entrada, o un pequeño número de entradas, en la tabla de página durante un periodo específico. Por ende, se puede usar una estructura parecida a caché, conocida como *translation lookaside buffer* (TLB, o memoria auxiliar para traducción de direcciones) con el fin de mantener un registro de las traducciones de dirección más recientes.

Cuando una dirección virtual se traduce a una dirección física, primero se consulta el TLB. Tal como en el caché de procesador, una porción del número de página virtual especifica dónde buscar en el TLB, y el resto se compara contra la tag de la(s) entrada(s) almacenada(s). Si las tags empatan y la entrada asociada es válida, entonces el número de página físico se lee y usa. De otro modo, se tiene un fallo de TLB, que se maneja en una forma similar a un fallo de caché de procesador. En otras palabras, se consulta la tabla de página, se obtiene la dirección física y el resultado de la traducción se graba en el TLB para uso futuro, posiblemente para sustituir una entrada TLB existente.

Por lo general, un TLB tiene decenas de miles de entradas, las más pequeñas son totalmente asociativas y las más grandes tienen menores grados de asociatividad. Un TLB constituye una memoria caché dedicada a entradas de tabla de página. Cuando se quiera sustituir una entrada TLB, sus *flags* se

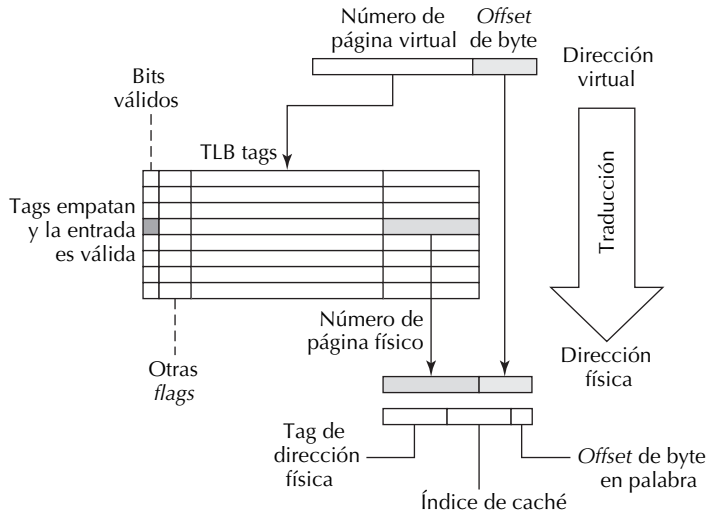


Figura 20.6 Traducción de dirección virtual a física mediante un *translation lookaside buffer*, que muestra cómo se usa la dirección física resultante para acceder a la memoria caché.

deben copiar en la tabla de página. Sin embargo, por fortuna, esto no implica mucha penalización de tiempo porque los fallos de TLB son muy raros. La figura 20.6 muestra el TLB y su uso para traducción de dirección.

Ejemplo 20.2: Traducción de dirección a través de TLB Un proceso particular de traducción de dirección convierte una dirección virtual de 32 bits a una dirección física de igual número de bits. La memoria es direccionable por byte y usa páginas de 4 KB. Se usa un TLB de mapeo directo y 16 entradas. Especifique los componentes de las direcciones física y virtual y el ancho de los diversos campos en el TLB.

Solución: El byte *offset* se toma 12 bits tanto en direcciones virtuales como físicas. Los 20 bits restantes en la dirección virtual (física) forman el número de página virtual (física). El número de página virtual de 20 bits se descompone en un índice de cuatro bits para abordar el TLB de 16 entradas y una tag de 16 bits; al último se compara contra la tag que se lee del TLB para determinar si hubo un impacto en el TLB. Además de la tag de 16 bits, cada entrada de TLB debe contener un número de página físico de 20 bits, un bit válido y otras *flags* sobre las que se copia desde la entrada de tabla de página correspondiente.

La discusión del TLB comenzó con una dirección virtual que se presentó a la memoria principal y mostró cómo se puede evitar el primer acceso a memoria para consultar la tabla de página. Falta por describir cómo funciona el TLB, o traducción de dirección virtual a física, en presencia de una caché de procesador. Hay dos enfoques para uso combinado de memoria virtual y un caché de procesador.

En una *caché de dirección virtual*, el caché está organizado de acuerdo con direcciones virtuales. Cuando una dirección virtual se presenta al controlador de caché, se divide en varios campos (*offset*, índice de línea/conjunto, tag virtual) y al caché se accede como siempre. Si el resultado es un impacto, no se necesita nada más. Para un fallo de caché, se requieren la traducción de dirección y el acceso a memoria principal. Por tanto, sólo se incurre en tiempo de acceso a TLB y otras cabeceras de traducción de dirección en el evento de un fallo de caché, que es un evento raro.

En una *caché de dirección física*, a TLB se accede antes que al caché. La dirección física que se obtiene del TLB, o de la tabla de página en el evento de un fallo de TLB, se presenta al caché, ello conduce a un impacto o fallo de caché. Para un fallo, a la memoria principal se accede con la dirección física que se obtuvo antes del acceso a caché. Puesto que el TLB es justo como un caché, con este enfoque se duplica el tiempo para ingresar datos en el caché, incluso ignorando los fallos TLB (que son raros). Sin embargo, el acceso a TLB se puede incorporar en una etapa de *encauzamiento* separada, de modo que el incremento de latencia produce poca o ninguna reducción en el rendimiento total (figura 17.10). El rendimiento total se reduce sólo en la medida en que el encauzamiento más profundo pueda requerir burbujas más frecuentes.

Dado que una *caché de dirección virtual* elimina la cabecera de traducción de dirección en la mayoría de los casos, se podría preguntar por qué se usan las cachés de dirección física. La respuesta es que, aun cuando las cachés de dirección virtual reduzcan la cabecera de traducción de dirección, llevan a complicaciones en otras partes. En particular, cuando los datos se deben compartir entre múltiples procesos (incluidos procesos de aplicación e I/O), esos datos pueden tener diferentes direcciones virtuales en los espacios de dirección de muchos procesos, lo anterior conduce a *aliasing* (efecto diente de sierra) o la existencia de múltiples copias (potencialmente inconsistentes) de los mismos datos en la caché.

Es posible usar una *caché de dirección híbrida* para ganar la rapidez de una caché de dirección virtual mientras se evitan sus problemas de *aliasing*. Un enfoque híbrido comúnmente usado es hacer la ubicación de datos en la caché una función sólo de los bits *offset* de página que no cambian durante la traducción de dirección. Por ejemplo, si el tamaño de la página es 4 KB (*offset* de 12 bits), y el tamaño de la línea caché es de 32 B (*offset* de cinco bits), habrán $12 - 5 = 7$ bits en la dirección virtual que son invariantes durante la traducción de dirección y se pueden usar para determinar la línea caché o conjunto caché en el que reside la palabra a que accede.

El método anterior, combinado con el uso de tags físicas en la caché conduce a un esquema híbrido, al que se puede referir como *virtualmente indexado, físicamente etiquetado*. Hasta la presentación de una dirección virtual, al caché de procesador y al TLB se accede en paralelo. Si la tag física recuperada del caché de procesador empata con la tag que se lee del TLB, entonces se pueden usar los datos de caché. De otro modo, se tiene un fallo de caché. Sin embargo, incluso en este caso el resultado de la traducción de dirección ya está disponible, a menos, desde luego, que se tenga un fallo de TLB. La figura 20.7 muestra en forma gráfica estas opciones de direccionamiento híbrido y puro.

Observe que con el caché de procesador, TLB y memoria virtual, un acceso de memoria puede conducir a fallos de tres tipos: fallo de caché, fallo de TLB y fallo de página. Sin embargo, no todas las ocho posibles combinaciones de impacto/fallo en estas tres entidades tienen sentido.

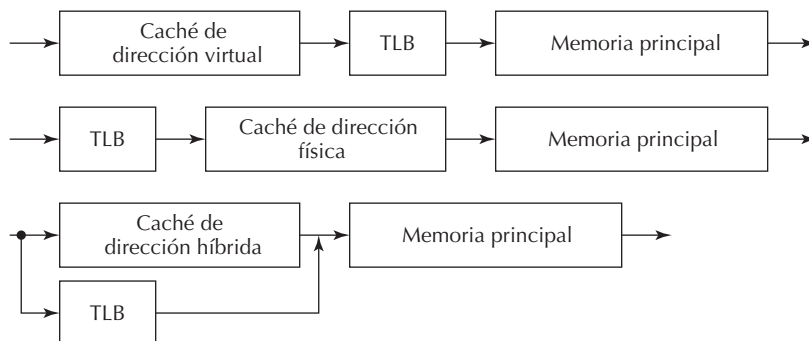


Figura 20.7 Opciones para donde ocurre la traducción de dirección virtual a física.

20.4 Políticas de sustitución de página

Con base en la discusión realizada hasta el momento, el diseñador de un sistema de memoria virtual debe resolver las siguientes preguntas clave, que fundamentalmente son las mismas que las encontradas para memorias caché:

1. *Política de lectura (fetch)*: cuándo llevar una página particular a la memoria principal.
2. *Política de colocación*: dónde colocar una página recientemente traída (y dónde encontrarla cuando se necesite).
3. *Política de sustitución*: cómo elegir entre las opciones de colocación (cuál de las páginas existentes ocupa actualmente aquellas localidades para sobrescribir).

La política de lectura (*fetch*) más común es la *paginación a petición*: traer una página hasta el primer acceso a ella. Una alternativa es la *prepaginación*, esto significa que las páginas se pueden traer antes de que las necesite un programa que corre. Lo anterior puede ser benéfico, por ejemplo, si cuando se accede a una página particular, también se traen las páginas cercanas en la misma pista de disco, pues hacer esto último incurre en muy poca penalización de tiempo, al menos en cuanto a acceso a disco se refiere. Puesto que los sistemas de disco más modernos emplean cachés especiales para retener sectores cercanos o incluso pistas completas hasta cada acceso a disco (sección 19.4), no se necesita considerar más prepaginación.

La naturaleza completamente asociativa del esquema de mapeo de página en memoria principal implica que la colocación de página no representa un problema (una página se puede colocar en cualquier parte en la memoria principal). Es esta colocación irrestricta la que necesita el uso de acceso indirecto a través de una tabla de página para ubicar una página, y el uso de un TLV para prevenir este acceso extra en la mayoría de los casos. En contraste, en los mapeos directo y de conjunto asociativo usados en cachés, la rendija sola, o un puñado de posibles localidades, para una línea de caché le permite encontrarse rápidamente mediante simples ayudas de hardware. En consecuencia, se ignorará por completo el conflicto de colocación en memoria virtual. No obstante, se nota que, en la mayoría de los sistemas de cómputo avanzados que tienen multitud de unidades de memoria local (rápido acceso) y remota (alta latencia) disponibles, la colocación no se convierte en conflicto.

El resto de esta sección se dedica a las políticas de *sustitución de página* para elegir una página a sobrescribir cuando no hay *marco de página* disponible en memoria principal para acomodar una página entrante. Puesto que la lectura (*fetch*) de una página desde disco toma millones de ciclos de reloj, el sistema operativo puede permitir el uso de una política de sustitución de página sofisticada que tome en cuenta la historia y otros factores. Es probable que el tiempo de corrido de incluso un proceso de decisión bastante complejo sea pequeño en comparación con los muchos milisegundos que se emplean en un solo acceso de disco. Esto último está en contraste con las políticas de sustitución en una caché de procesador o TLB, que debe ser en extremo simple.

Idealmente, una política de sustitución de página minimiza el número de fallas de página durante el curso de la ejecución de un programa. Tal política ideal no se puede implementar porque requeriría conocimiento perfecto acerca de todos los accesos futuros de página. Las políticas de sustitución prácticas están diseñadas para acercarse a una política ideal con el uso del conocimiento acerca del comportamiento pasado de un programa para predecir su curso futuro. El algoritmo de *usado menos recientemente* (LRU) es uno de tales políticas. Implementar la política LRU es difícil; requiere que cada página se marque en el tiempo con cada acceso y que la marca de tiempo se compare para encontrar el más pequeño entre ellas al momento de sustitución. Por esta razón, con frecuencia se implementa una versión aproximada de la política LRU, que funciona muy bien en la práctica.

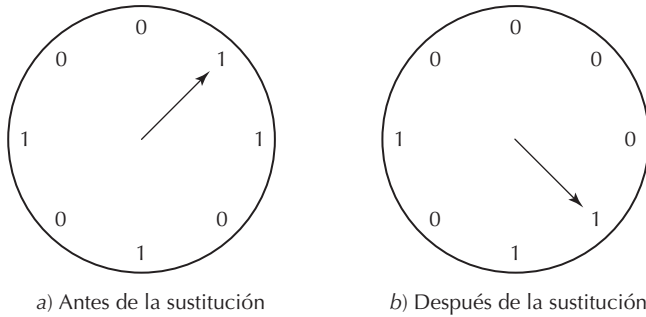


Figura 20.8 Esquema de la implementación aproximada de LRU.

Una forma de implementar LRU en una forma aproximada se describe a continuación. Un *bit de uso* se asocia con cada página; dicho bit especifica si se realizó acceso a dicha página en el pasado reciente (desde el último restablecimiento del bit de uso por parte del sistema operativo). El bit de uso en esencia divide las páginas en dos grupos: con y sin acceso recientes. Sólo las páginas en la última categoría son candidatos para sustitución. Se pueden usar muchos métodos para seleccionar entre los candidatos y para restablecer periódicamente los bits de uso. En la figura 20.8 se muestra uno de tales esquemas, al que a veces se le refiere como *política de reloj*, donde los números 0 y 1 son bits de uso para las diversas páginas, y la flecha es un apuntador que marca una de las páginas. Cuando se debe seleccionar una página para sustitución, la flecha se mueve en sentido de las manecillas del reloj, restablece a 0 todos los 1 que encuentra y se detiene en el primer 0, que es donde se colocará la nueva página, con su bit de uso fijado a 1. Una página cuyos bits de uso se cambian de 1 a 0 se sustituirá cuando la flecha haga una rotación completa, a menos, desde luego, que se use de nuevo antes y su bit de uso se fije a 1. Se ve que las páginas con accesos frecuentes nunca se sustituirán bajo esta política.

La memoria virtual funciona porque saca toda la ventaja tanto de la localidad espacial como de la temporal de los accesos a memoria en los programas típicos. La *localidad espacial* justifica llevar miles de bytes a la memoria principal cuando se encuentra una falla de página durante un intento por acceder un solo byte o palabra. Debido a la localidad espacial, mucho del resto de la página quizá se usará en lo futuro, más de una vez, ello amortiza la latencia de un solo acceso a disco durante muchas referencias de memoria. La *localidad temporal* hace exitosa la política de sustitución LRU para mantener las instrucciones y datos más útiles (desde el punto de vista de referencias futuras de memoria) en la memoria principal y sobrescribir las partes menos útiles. Si una instrucción o elemento de datos no se usa durante algún tiempo, hay posibilidades de que el programa se mueva para hacer otras cosas y no necesitará dicho objeto en lo futuro. Desde luego, no es difícil escribir un programa que no muestre

Ejemplo 20.3: No siempre LRU es la mejor política Considere el siguiente fragmento de programa que trata con una tabla *T* con 17 hileras y 1024 columnas, calcula un promedio para cada columna de tabla y lo imprime a la pantalla (*i* es el índice de hilera y *j* el índice de columna):

```
for j = [0 ... 1023] {
    temp = 0;
    for i = [0 ... 16]
        temp = temp + T[i][j]
    print(temp/17.0); }
```

T[i][j] y *temp* son valores punto flotante de 32 bits y la memoria es direccionable por palabra. La variable temporal *temp* se mantiene en un registro de procesador, de modo que acceder a *temp* no implica una

referencia de memoria. La memoria principal está paginada y retiene 16 páginas de 1024 palabras de tamaño. La política de sustitución de página es “menos recientemente usada”.

- Si supone que T se almacena en el espacio de dirección virtual en formato de hilera mayor, ¿cuántas fallas de página se encontrarán y cuál es la razón de impacto de memoria principal?
- ¿Qué fracción de las fallas de página en la parte a) es obligatoria? ¿De capacidad? ¿De conflicto?
- Repita la parte a), pero esta vez suponga que T se almacena en el formato de columna mayor.
- ¿Qué fracción de las fallas de página de la parte c) es obligatoria? ¿De capacidad? ¿De conflicto?

Solución: La figura 20.9 ofrece una visualización de cómo se dividen los elementos de T entre 17 páginas para los dos casos de almacenamiento de hilera y columna mayor.

- Después de llevar la hilera 0 a la memoria principal, se accede a otras 16 hileras antes de que la hilera 0 se necesite de nuevo. ¡La política LRU fuerza a cada página a salir antes de su siguiente uso! Por tanto, cada acceso causa una falla de página. Existe un total de $1024 \times 17 = 17408$ fallos de página y la razón de impacto es 0.
- Puesto que sólo hay 17 páginas, 17 o casi 0.1% de los fallos se pueden considerar obligatorios. No hay fallos de conflicto con un esquema de mapeo completamente asociativo. Los restantes 17391, o 99.9% de los fallos lo son de capacidad.
- Cada página contiene $1024/17 \cong 60$ columnas de T. Una página nunca se usa de nuevo cuando sus columnas se procesan. Por tanto, sólo hay 17 fallos y la razón de impacto es 99.9%.
- Los 17 fallos son obligatorios; no hay fallos de capacidad o de conflicto.

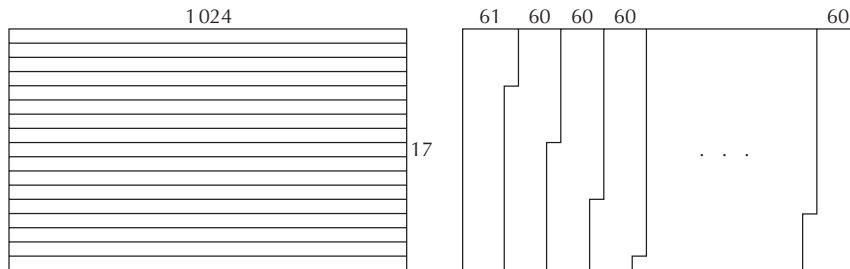


Figura 20.9 Paginación de una tabla de 17×1024 cuando se almacena en orden de hilera mayor o columna mayor.

localidad espacial ni temporal. Tal programa hará que la memoria virtual tenga mal rendimiento. Pero la mayoría del tiempo lo hará muy bien.

Es posible que una página se haya modificado desde que se llevó a la memoria principal. Por ende, para asegurar que no se pierde información, las páginas se deben copiar de vuelta a disco antes de sobrescribir con nuevas páginas entrantes. En virtud de que esta reescritura consume tiempo, un “bit sucio” se asocia con cada página que se fija, siempre que ésta se modifique. Si una página no está sucia, no se necesita reescritura y se ahorra tiempo. Observe que, debido a la extrema penalización de tiempo del acceso a disco, aquí no es opción el uso de esquema de escritura a través. Sin embargo, con reescritura, el número de accesos a disco se minimiza y la cabecera se vuelve aceptable, como lo fue entre caché y memoria principal. Para evitar agregar la penalización de escritura y al largo tiempo de acceso a disco hasta una falla de página, el sistema operativo puede preseleccionar páginas para sustituir e iniciar el proceso de escritura antes de que surja una necesidad real. Adicionalmente, el sistema operativo puede encerrar ciertas páginas útiles o frecuentemente necesarias de modo que nunca se sustituyan.

La siguiente analogía puede ser útil. Usted tiene libreros de capacidad limitada para sus libros técnicos y decide comprar o conservar un nuevo libro técnico sólo si puede descartar un libro viejo que ya no le sea útil. Ciertos libros son muy útiles en su trabajo y los marca como no sustituibles. Si descarta libros por adelantado, es probable que tenga espacio disponible para nuevos libros cuando lleguen; de otro modo, puede verse forzado a apilar libros en su escritorio hasta que encuentre tiempo para decidir cuáles libros descartar. Con frecuencia, ese tiempo nunca llega, lo cual hace que la pila en su escritorio sea cada vez más alta, hasta que se derrumba y lo lastima con la perturbación más ligera.

■ 20.5 Memorias principal y masiva

La memoria virtual no sólo hace posible que un proceso se expanda más allá del tamaño de la memoria principal disponible, también permite que múltiples procesos compartan la memoria principal con gestión de memoria ligeramente simplificada. Al encontrar una falla de página, un proceso se coloca en estado latente y el procesador se asigna a alguna otra actividad. En otras palabras, una falla de página se trata como una excepción que dispara un interruptor de contexto. El proceso latente más tarde se activa cuando su página requerida ya está en la memoria principal. De esta forma, se puede transferir el control continuamente entre muchos procesos, cada uno queda latente si encuentra una falla de página u otra situación que requiera esperar la disponibilidad de un recurso. Cuando muchos procesos están disponibles para ejecución, el procesador siempre puede encontrar algo útil que hacer durante los largos periodos de espera creados por las fallas de página. En este sentido, poner más procesos en memoria principal significa que cada uno obtiene menos páginas, ello conduce a fallas de página frecuentes y a menor rendimiento. En consecuencia, se debe fijar un equilibrio entre reducir el tiempo de inactividad del procesador mediante la disponibilidad de procesos más activos y tener suficientes páginas de cada proceso en memoria principal para asegurar una tasa baja de falla de página.

Una noción útil en este aspecto es la de *conjunto operativo* (*working set*) de un proceso. Conforme el programa se ejecuta, sus accesos a memoria tienden a concentrarse en pocas páginas durante algún tiempo. Entonces, algunas de éstas se abandonan y el foco de interés cambia hacia otras. El conjunto operativo de duración x para un proceso representa el conjunto de todas las páginas a las que se ha hecho referencia en el curso de la ejecución de las últimas x instrucciones. El conjunto operativo cambia con el tiempo, de modo que se le denota como $W(t, x)$. El principio de localidad garantiza que $W(t, x)$ permanezca más bien pequeño para valores grandes de x . La figura 20.10 muestra variaciones típicas en el tamaño del conjunto operativo de un programa con el tiempo, donde los picos estrechos corres-

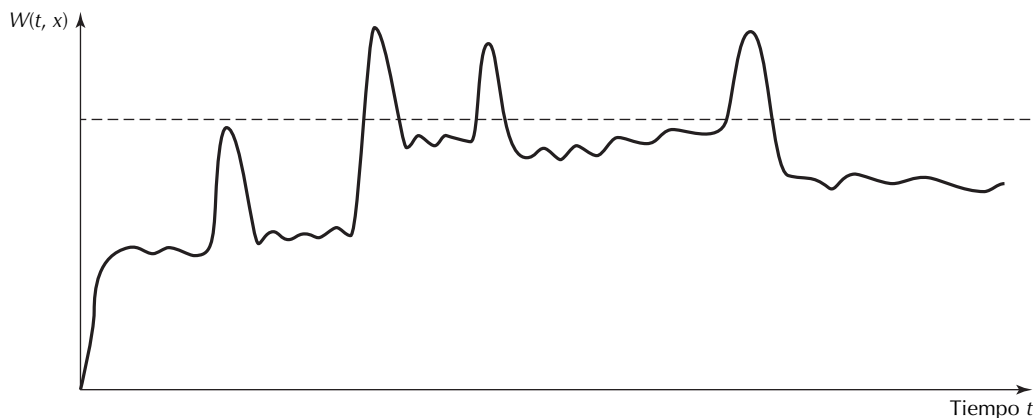


Figura 20.10 Variaciones en el tamaño del conjunto operativo de un programa.

ponden a corrimientos en las referencias de memoria del programa de una a otra localidad (durante los corrimientos, tanto a la localidad antigua como a la localidad nuevo se ha ingreso en el pasado reciente).

Si el conjunto operativo de un proceso reside en la memoria principal, entonces se encuentran pocas fallas de página. En este punto, asignar más memoria al proceso puede no tener un efecto significativo sobre el rendimiento. Si la línea punteada de la figura 20.10 representa la cantidad de memoria principal asignada al programa, entonces el programa correrá sin una falla de página durante largos periodos.

Si se conoce exactamente cuál sería el conjunto operativo de un proceso en un momento dado, se podrían *preleer* sus miembros en la memoria principal y se evitaría la mayoría de las fallas de página. Desafortunadamente, dado que el comportamiento pasado de un programa no siempre refleja con precisión lo que ocurrirá en lo futuro, esto es imposible de hacer. De este modo, la noción de conjunto operativo con frecuencia se usa como herramienta conceptual en asignación de memoria. Si se vigilan las fallas de página generadas por un proceso, se obtiene una indicación bastante precisa de si la cantidad de memoria asignada al proceso es suficiente para acomodar su conjunto operativo. Las fallas de página frecuentes indican que la memoria asignada es inadecuada. Si un proceso debe suspenderse para mejorar el rendimiento de memoria, el hecho de que se produzcan las mayores fallas de página constituye un buen signo. En este orden de ideas, si todos los procesos activos producen un bajo nivel de fallas de página, entonces quizás se pueda activar un nuevo proceso y llevarlo a la memoria principal.

Por lo general, los procesos se crean al colocar sus páginas en la memoria secundaria. Cuando va a comenzar la ejecución de un proceso, se le asignan cierto número de páginas de memoria principal y gradualmente se llevan sus páginas (paginación a petición). Algunas de estas páginas, que se espera tengan accesos frecuentes, se pueden cerrar en la memoria principal. Cuando las páginas inicialmente asignadas se usan, están disponibles muchas opciones hasta que se encuentra la siguiente falla de página. Con una *política de sustitución local*, se elige para sustitución una de las páginas no cerradas del proceso. Esto último propicia que el número de páginas de memoria principal asignadas al proceso permanezca constante con el tiempo. Una *política de sustitución global* elige la página a sustituir sin importar a cuál proceso pertenezca. Esto causará que más procesos activos, o los que tengan conjuntos operativos más grandes, ocupen gradualmente más páginas a costa de otros procesos; por tanto, se conduce a una forma de asignación y equilibrio automáticos de memoria; conforme un proceso deja de usar ciertas páginas, puede ceder espacio a otros procesos que sean más activos, y luego recuperar algún espacio conforme produzca más actividad por sí mismo.

■ 20.6 Mejora del rendimiento de la memoria virtual

La implementación y rendimiento de la memoria virtual son afectados por algunos parámetros cuyo impacto e interrelación deben entender bien no sólo los diseñadores, sino también los usuarios. Éstos incluyen:

- Tamaño de memoria principal.
- Tamaño de página.
- Política de sustitución.
- Esquema de traducción de dirección.

Los primeros tres parámetros y sus interrelaciones son muy similares a las de las cachés de procesador. Por tanto, en la tabla 20.1 se citan los parámetros relacionados con esquemas de memoria jerárquica (de las que son ejemplos las memorias caché y virtual) y sus efectos de rendimiento. Ya está familia-

■ **TABLA 20.1** Parámetros de jerarquía de memoria y sus efectos sobre el rendimiento.

Variación de parámetro	Ventajas potenciales	Posibles desventajas
Mayor tamaño principal o caché	Menos fallos de capacidad	Tiempo de acceso más largo
Páginas o líneas más grandes	Menos fallos obligatorios (efecto de <i>prefetching</i>)	Mayor penalización por fallo
Mayor asociatividad (sólo para caché)	Menos fallos de conflicto	Tiempo de acceso más largo
Política de sustitución más sofisticada	Menos fallos de conflicto	Tiempo de decisión más largo, más hardware
Política de escritura (sólo para caché)	No penalización por tiempo de reescritura, manipulación más sencilla de fallo de escritura	Ancho de banda de memoria desperdiciado, tiempo de acceso más largo

rizado con las afirmaciones de la tabla 20.1 en relación con cachés. A continuación se justificará lo mismo para la memoria virtual y se discutirán más conflictos relacionadas con la implementación de tablas de página para traducción de dirección.

El impacto en el rendimiento del tamaño de la memoria principal es obvio. en el extremo, cuando la memoria principal es muy grande como para retener todo un programa y sus datos, sólo se tienen fallos obligatorios. Sin embargo, una memoria principal muy grande no sólo es más costosa sino también más lenta. De modo que hay una penalización por rapidez oculta que se debe considerar, junto con los conflictos costo/efectividad al elegir el tamaño de una memoria principal.

La elección de un tamaño de página es función del comportamiento típico del programa, así como de la diferencia en las latencias y tasas de datos de las memorias principal y masiva. Cuando los accesos a programa tienen gran cantidad de localidad espacial, un tamaño de página más grande tiende a reducir el número de fallos obligatorios debido al efecto de *prefetching*. No obstante, las mismas páginas más grandes desperdiciarán espacio y ancho de banda de memoria cuando haya poca o ninguna localidad espacial. Así pues, con frecuencia existe un tamaño de página óptimo que conduce al mejor rendimiento promedio para programas que usualmente “corren” en una computadora específica. Mientras mayor sea la razón del tiempo de acceso a disco a latencia de memoria principal, mayor será el tamaño de página óptimo. La figura 20.11 muestra cómo el tiempo de búsqueda de disco (que varía casi en proporción a la latencia de disco total) y el tiempo de acceso a memoria principal cambiaron con los años. Se ve que la razón de los dos parámetros permaneció constante. Este hecho, junto con memorias caché más rápidas que efectivamente ocultan alguna de la latencia DRAM, explica por qué los tamaños de página han rondado los 4 KB.

Es interesante observar que, a principios de la década de 1980, conforme la brecha de rapidez entre memorias de disco y DRAM parecía destinada a crecer (mientras que sus densidades de almacenamiento se acercaban más), las predicciones de que los discos gradualmente darían su lugar a memorias de semiconductor eran un lugar común. En parte, como resultado de este reto, los fabricantes de discos intensificaron sus investigaciones y esfuerzos de desarrollo y tuvieron éxito al reducir realmente la brecha en la década siguiente (figura 20.11).

El impacto de la política de sustitución sobre el rendimiento se ha estudiado ampliamente, y muchas políticas novedosas y refinamientos a los métodos antiguos han ofrecido un esfuerzo por reducir la frecuencia de fallas de página. La figura 20.12 muestra los resultados experimentales sobre la tasa de fallas de página de LRU y LRU aproximada, en comparación con el algoritmo ideal y el primer algoritmo *first-in, first-out* (FIFO: primero en entrar, primero en salir), esto se basa en un estudio muy antiguo realizado al correr un programa Fortran con un tamaño de página fijo de 256 palabras [Baer80]. Sin embargo, otros investigadores han notificado resultados similares en diferentes contextos. El número de páginas asignadas a un proceso se muestra junto al eje horizontal y se puede ver como la representa-

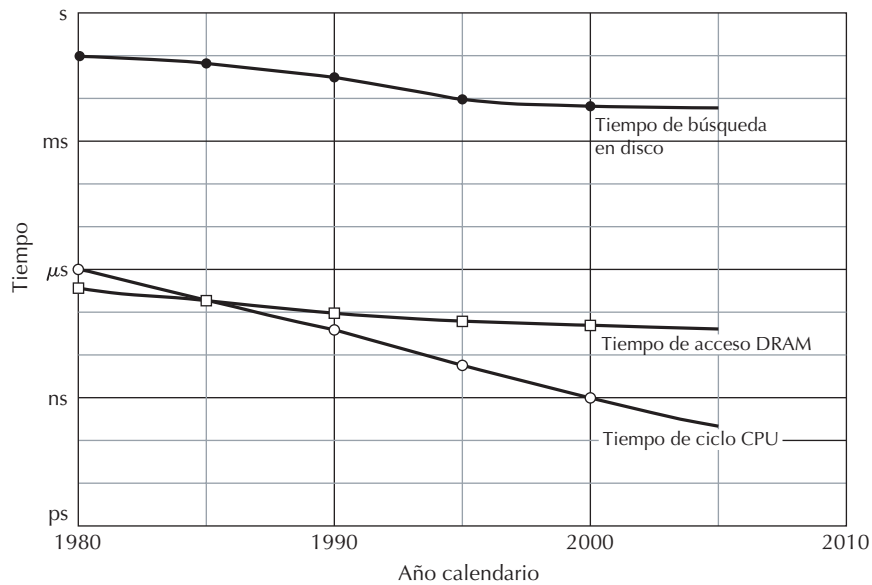


Figura 20.11 Tendencias en rapidez de disco, memoria principal y CPU.

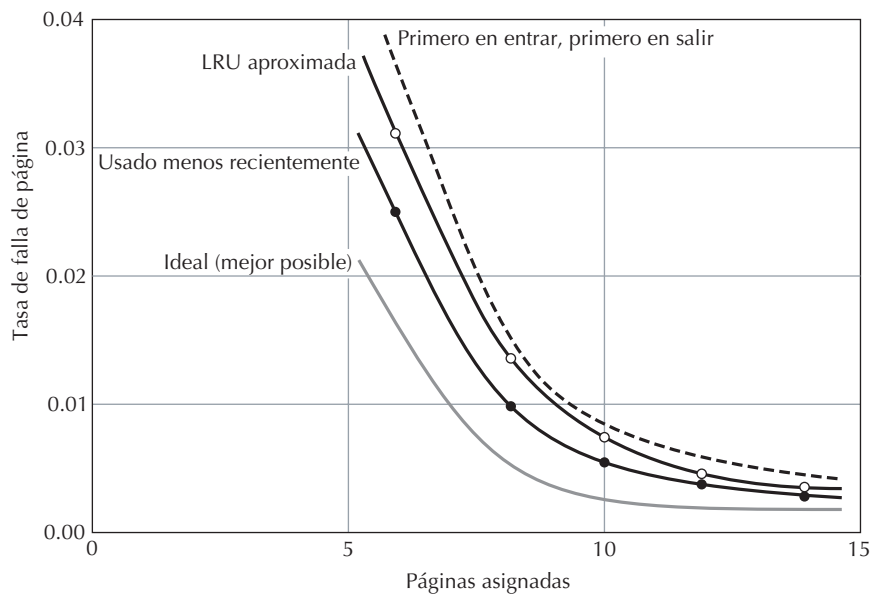


Figura 20.12 Dependencia de las fallas de página sobre el número de páginas asignadas y la política de sustitución de página.

ción del tamaño de la memoria principal en un entorno de un solo proceso. Con base en las tendencias mostradas en la figura 20.12, se concluye que el efecto relativo de la política de sustitución sobre la tasa de falla de página está confinado a la vecindad de un pequeño factor constante (alrededor de 2). La diferencia es muy significativa en términos absolutos cuando un pequeño conjunto de páginas se asigna a un proceso (o, de modo equivalente, la memoria principal es más bien limitada), y la diferencia se vuelve menos pronunciada cuando la memoria está llena.

Conceptualmente, la traducción de dirección se implementa a través de una tabla de página y su impacto en el rendimiento se suaviza con el uso de un TLB. Como se observó antes, las tablas de páginas pueden ser enormes. Por ejemplo, las direcciones virtuales de 32 bits implican tablas de página de 1 M de entradas cuando el tamaño de la página es de 4 KB. Desde luego, la tabla de página debe residir en la memoria principal si se debe realizar rápidamente la traducción de dirección requerida en el caso de un fallo de TLB. Esto último se logra sin tener que separar para las tablas de página grandes áreas de memoria principal poco usadas. El truco es dejar que las páginas de la tabla de página se barran de la memoria como todas las otras páginas. El principio de localidad aseguraría que los “trozos” más útiles de la tabla de página permanecerían en memoria principal y se podría acceder a ellos rápidamente.

Una alternativa para tener una gran tabla de página monolítica consiste en usar una estructura de dos niveles. Por ejemplo, para números de página virtual de 20 bits, los diez bits superiores se podrían usar como índice en un directorio de página (tabla de página nivel 1) con 1 024 entradas. Cada entrada en este directorio de página contiene un apuntador hacia una tabla de nivel 2 de 1 024 entradas. Con entradas de 32 bits, cada una de tales tablas encaja en una página de 4 096 B. El directorio de página puede cerrarse en memoria principal y barrer las tablas de nivel 2, según se requiera. Una ventaja de este esquema es que para programas típicos que usan algunas páginas contiguas al comienzo de su espacio de dirección virtual y una o más páginas al final para retener una pila, sólo se necesitarán alguna vez pocas tablas de nivel 2, ello hace posible que éstas residan en la memoria principal. Un inconveniente de las tablas de página de dos niveles es la necesidad de tener dos accesos a tabla por traducción de dirección. Sin embargo, el uso de un TLB mitiga lo anterior en gran medida.

PROBLEMAS

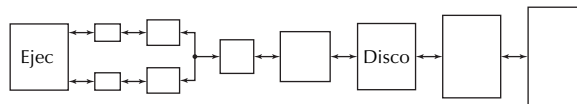
20.1 Analogías de paginación en la vida cotidiana

Describa qué tiene en común cada una de las situaciones siguientes con la memoria virtual y la paginación, elabore sobre las contrapartes de lectura (*fetch*), colocación y políticas de sustitución, así como en traducción de dirección, donde sea aplicable.

- Gerente de supermercado que asigna espacio de anaqueles a productos.
- Películas que se eligen para proyectar en cines de todo el mundo.
- Automóviles estacionados en una calle donde hay parquímetros.

20.2 Partes de una jerarquía de memoria

Considere el siguiente diagrama de un sistema de memoria jerárquica:



- Asigne una etiqueta adecuada a cada recuadro sin etiqueta y explique su función.
- Mencione tecnologías de hardware útiles para cada nivel.
- Proporcione un valor aproximado para las razones de latencia entre niveles sucesivos.
- Proporcione una cifra aproximada para la razón de impacto en cada nivel.
- Mencione esquemas de colocación y sustitución típicos en cada nivel.
- Describa el mecanismo de traducción de dirección, si hubiera alguno, en cada nivel.

20.3 Conceptos de memoria virtual

Responda cierto o falso, con justificación completa.

- La diferencia numérica entre una dirección virtual y la correspondiente dirección física siempre es distinta de cero.
- El tamaño de un espacio de dirección virtual puede exceder la capacidad total de disco en bytes.
- No es posible correr un mecanismo de traducción de dirección hacia atrás, y obtener una correspondiente dirección virtual a partir de una dirección física.
- Las páginas se deben hacer tan grandes como sea posible para minimizar el tamaño de las tablas de página.
- Cuando muchos procesos comparten una página, sus correspondientes entradas de tabla de página deben ser idénticas.
- Una política de sustitución óptima que siempre conduce a menores fallas de página posibles no es útil.

20.4 Direcciones virtual y física

Un sistema de memoria virtual direccionable por byte con páginas de 4096 B tiene, en memoria principal, marcos de ocho páginas virtuales y cuatro páginas físicas. Las páginas virtuales 1, 3, 4 y 7 residen en los marcos de página 3, 0 2 y 1, respectivamente.

- Especifique los rangos de dirección virtual que conducirían a fallas de página.
- Encuentre las direcciones físicas que correspondan a las direcciones virtuales 5000; 16200; 17891 y 32679.

20.5 Movimiento de datos en jerarquía de memoria

- Dibuje y etiquete adecuadamente todos los componentes de un diagrama similar al de la figura 20.2 que muestre dos niveles de *caching*.
- Repita la parte a), pero esta vez suponga *caching* de un solo nivel con cachés divididas.
- Dibuje un diagrama que combine las partes a) y b); esto es, cachés dividida L1 y unificada L2.

20.6 Traducción de dirección a través de TLB

- Muestre cómo funcionaría cada una de las opciones de traducción en la figura 20.7 con el TLB especificado en el ejemplo 20.2.

- Vuelva a hacer el ejemplo 20.2 con un TLB de conjunto asociativo de dos vías y 64 entradas.
- Repita la parte a) para el TLB especificado en la parte b).

20.7 Algoritmo de sustitución LRU aproximado

¿Cómo se comportaría el fragmento de programa del ejemplo 20.3 bajo el algoritmo de sustitución LRU aproximada? Establezca claramente todas sus suposiciones.

20.8 Algoritmo de sustitución MRU

Un amigo le sugiere que la sustitución de la página *usada más recientemente* (MRU) podría ser una alternativa viable. Usted intenta convencerlo de que no es buena idea, dadas las propiedades de localidad espacial y temporal de los programas. Él insiste que MRU puede reemplazar LRU en algunos casos.

- ¿La afirmación de su amigo tiene algún mérito?
- ¿MRU es más fácil de implementar que LRU?

20.9 Algoritmo de sustitución LRU aproximada

La figura 20.8 ejemplifica la aplicación de los “bits de uso” en la implementación de una versión aproximada del algoritmo LRU.

- ¿Bajo qué condiciones esta versión aproximada muestra exactamente el mismo comportamiento que LRU?
- Una variación de este método también usa “bits sucios” (conocidos como “bits modificados”) en la toma de decisión de sustitución. Describa cómo podría funcionar esta forma más sofisticada del algoritmo.
- ¿Cuáles son los beneficios de usar un “contador de uso” de dos bits en lugar de un solo “bit de uso” para este algoritmo?

20.10 Conjuntos operativos

- Si ignora el enunciado print, convierta el fragmento de programa del ejemplo 20.3 en una secuencia de instrucciones MiniMIPS.
- Grafique las variaciones del conjunto operativo para el programa de la parte a) con base en las últimas 20,

40, 60, . . . instrucciones y suponga que la tabla se almacena en formato de hilera mayor.

- c) Repita la parte b), pero esta vez suponga que la tabla se almacena en formato de columna mayor.

20.11 Frecuencia de falla de página

Considere la ejecución del siguiente ciclo, donde los arreglos A, B y C están almacenados en páginas separadas de 1 024 palabras:

```
for i in 0 ... 1023 do
    C[i] := A[i] + B[i]
endfor
```

Indique la secuencia de referencias de página y discuta la frecuencia de fallas de página con varias cantidades de memoria principal asignada al programa que contiene el bucle. Establezca claramente todas sus suposiciones.

20.12 Política de sustitución de página

Un proceso accede a sus cinco páginas en el orden siguiente: A B C D A B E A B C D E.

- Determine cuáles accesos de página causan una falla de página si hay tres marcos de página en memoria principal y la política de sustitución es FIFO, LRU o LRU aproximada (tres casos).
- Repita la parte a) para marcos de cuatro páginas.

20.13 Política de sustitución de página óptima

Muestre que se obtiene una política de sustitución óptima si se sustituye la página cuya siguiente referencia es la más lejana en lo futuro [Matt70].

20.14 Tiempo de acceso promedio a memoria principal

En un sistema de paginación a petición, el tiempo de ciclo de memoria principal es de 150 ns y el tiempo de acceso promedio a disco es de 15 ms. La traducción de dirección requiere un acceso a memoria en el caso de un fallo de TLB y tiempo perdido de otro modo. La tasa de impacto TLB es 90%. De los fallos TLB, 20% conducen a una falla de página (2% de los accesos totales). Calcule un tiempo de acceso promedio para

este sistema de memoria virtual. ¿En qué forma es engañoso el tiempo de acceso promedio calculado?

20.15 Reducción de las fallas de página mediante bloqueo

El ejemplo 20.3 mostró que, dependiendo de si una tabla se almacena en orden de hilera mayor o columna mayor, se generan fallas de página excesivas o muy pocas fallas de página. Considere el problema de la multiplicación de matriz con matrices cuadradas de 256×256 . Suponga una asignación de memoria principal total de 64 páginas, donde cada una contiene 1 024 palabras, a las dos matrices operando y la matriz resultado.

- Si supone almacenamiento en hilera mayor de las tres matrices involucradas, derive el número de fallas de página cuando la multiplicación de matriz se realiza vía el algoritmo simple con tres ciclos anidados al calcular c_{ij} como la suma de los términos producto $a_{ik} \times b_{kj}$ sobre todos los valores de k y se usa el algoritmo de sustitución LRU.
- Repita la parte a) con almacenamiento de columna mayor de las tres matrices.
- Demuestre que si las matrices se visualizan como bloques de 32×32 y la multiplicación de matriz se realiza en las matrices resultantes de bloques 8×8 , el número de fallas de página se reduce significativamente.

20.16 Tablas de página de dos niveles

Como se mencionó al final de la sección 20.6, las tablas de página de dos niveles se usan para reducir los requisitos de memoria de una gran tabla de página monolítica que usualmente tiene grandes bloques de entradas no usadas.

- Dibuje un diagrama similar al de la figura 20.4, pero con una tabla de página de dos niveles.
- Repita la parte a) para la figura 20.5.
- ¿Cómo se organiza el TLB cuando se tienen tablas de páginas de dos niveles?
- Explique por qué tiene sentido que las tablas de página de segundo nivel ocupen una página cada una.
- Estudie la estructura de tabla de página de dos niveles de Pentium de Intel y prepare un resumen de dos páginas de su esquema de traducción de dirección, incluida la forma en que usa el TLB.

20.17 Rendimiento de memoria virtual

Dada una secuencia de números que representan números de página virtual en el orden en que se acceden, usted puede evaluar el rendimiento de un sistema de memoria virtual con parámetros conocidos al estimar cuáles accesos conducen a fallas de página. Considere ahora invertir la secuencia dada de modo que los acce-

sos a las páginas virtuales ocurren exactamente en el orden opuesto (por decir, C B B A A C B A en lugar de A B C A A B B C). ¿Puede deducir algo acerca del número de fallas de página que se encuentran con este ordenamiento inverso en relación con la secuencia de acceso original? Justifique completamente cualquier suposición que haga y su conclusión.

REFERENCIAS Y LECTURAS SUGERIDAS

-
- [Baer80] Baer, J. L., *Computer Systems Architecture*, Computer Science Press, 1980.
 - [Brya03] Bryant, R. E. y D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, Prentice Hall, 2003.
 - [Denn70] Denning, P. J., "Virtual Memory", *ACM Computing Surveys*, vol. 2, núm. 3, pp. 153-189, septiembre de 1970.
 - [Hand93] Handy, J., *The Cache Memory Book*, Academic Press, 1993.
 - [Henn03] Hennessy, J. L. y D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 3a. ed., 2003.
 - [Kilb62] Kilburn, T. D., B. G. Edwards, M. J. Lanigan y F. H. Sumner, "One-Level Storage System", *IRE Trans. Electronic Computers*, vol. 11, pp. 223-235, abril de 1962.
 - [Matt70] Mattson, R. L., J. Gecsei, D. R. Slutz y I. L. Traiger, "Evaluation Techniques for Storage Hierarchies", *IBM Systems J.*, vol. 9, núm. 2, pp. 78-117, 1970.
 - [Silb02] Silberschatz, A., P. B. Galvin y G. Gagne, *Operating Systems Concepts*, Wiley, 6a. ed., 2002.

PARTE SEIS

ENTRADA/SALIDA E INTERFASES

“Si pones basura en una computadora no sale sino basura. Pero esta basura, al haber pasado por una máquina muy costosa, se ennoblece un poco y nadie osa criticarla.”

Fuente desconocida (de los días cuando no se podía imaginar)

“Sabía que teníamos servicio telefónico mediante la televisión por cable y que transmitíamos nuestros documentos por medio de nuestra copiadora, pero ¿cuándo comenzamos a enviar faxes a través del tostador?”

Leyenda en una caricatura de artista desconocido

TEMAS DE ESTA PARTE

21. Dispositivos de entrada/salida
22. Programación de entrada/salida
23. Buses, ligas e interfases
24. Conmutación contextual e interrupciones

Muchos de los datos que se procesan con una computadora se originan fuera de la máquina, los resultados del cálculo se obtienen mediante la ejecución de programas que con mucha frecuencia van hacia fuentes externas de interpretación, archivado o mayor procesamiento. En este contexto, la latencia de las operaciones de entrada y salida puede tener un impacto significativo sobre la percepción del rendimiento de una computadora; importa poco que un CPU corra un programa en una fracción de segundo si tarda muchos minutos en proporcionar los datos que deben procesarse y obtener de vuelta los resultados. Por tanto, es imperativo, para diseño y uso efectivo de computadora, que se entienda cómo funcionan los dispositivos de entrada/salida, cómo interactúan con la memoria y el CPU, y cómo se puede usar una computadora para recolectar datos desde fuentes externas o para activar y controlar dispositivos externos.

En el capítulo 21 se revisan los tipos de dispositivos de entrada/salida y sus detalles de implementación que tienen un vínculo con el costo, el rendimiento o las tasas de datos requeridas para apoyar sus interacciones con la memoria y el CPU. Luego, en el capítulo 22, se examina el tema de la programación I/O y las implicaciones costo/rendimiento de enfoques alternos como el *polling* (sondeo), las interrupciones y DMA. Los buses y ligas (*links*) de otros tipos son los mecanismos primarios para

conectar componentes de computadora, mutuamente y hacia dispositivos periféricos. Dichos mecanismos y los estándares asociados, se discuten en el capítulo 23. El capítulo 24 trata las interrupciones que forman importantes mecanismos para implementar eficientes procesos entrada/salida asíncronos y también conduce a métodos para conmutar (*switching*) entre hilos de computación con la intención de evitar tiempos inactivos debidos a dependencias de datos y retardos de acceso.

■ CAPÍTULO 21

DISPOSITIVOS DE ENTRADA/SALIDA

“Cuando el usuario construye un sendero, le da nombre, inserta éste en su libro de códigos y lo teclea en su teclado... Así se cree que los objetos físicos se reúnen de fuentes muy separadas y se enlazan para formar un libro nuevo. Es más que esto, pues cualquier objeto se puede unir en numerosos senderos.”

Vannevar Bush, Como podemos pensar, 1945 (donde profetiza lo que ahora es la World Wide Web)

“El valor de un programa es proporcional al peso de su salida.”

Sexta ley de la programación de computadoras

TEMAS DEL CAPÍTULO

- 21.1** Dispositivos y controladores de entrada/salida
- 21.2** Teclado y ratón
- 21.3** Unidades de presentación visual
- 21.4** Dispositivos de entrada/salida de copia impresa
- 21.5** Otros dispositivos de entrada/salida
- 21.6** Redes de dispositivos de entrada/salida

En este capítulo se revisarán la estructura y principios operativos de algunos dispositivos comunes de entrada/salida, tanto dispositivos que ofrecen o presentan datos como los que capturan datos para archivar o para su posterior procesamiento. Además del tipo de presentación o grabación de datos, los dispositivos I/O se pueden categorizar por sus tasas de datos, desde dispositivos de entrada de datos muy lentos (teclados) hasta subsistemas de almacenamiento de gran ancho de banda. Se verá que la tasa de datos requerida puede dictar la forma en que interactúan el CPU, la memoria y los dispositivos I/O. Cada vez más, la fuente de entrada, o recipiente de salida, de una computadora es otra computadora que se liga a ella a través de una red. Por esta razón, la creación de redes (*networking*) de dispositivos I/O también se discute en este capítulo.

■ 21.1 Dispositivos y controladores de entrada/salida

El procesador y la memoria son mucho más rápidas que la mayoría de los dispositivos I/O. En general, mientras más cerca esté una unida al CPU, más rápido se accede a ella. Por ejemplo, si la memoria principal está a 15 cm de distancia del CPU (figura 3.11a), sólo la propagación de señal toma 1 ns en cada vía, pues las señales electrónicas viajan casi a la mitad de la rapidez de la luz; a este retardo de

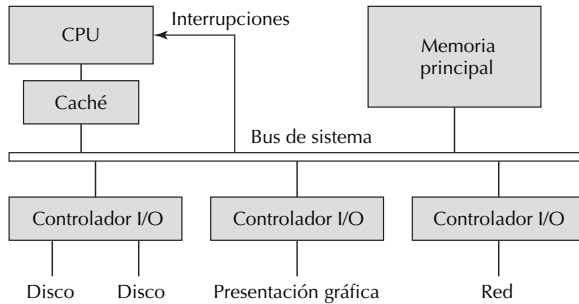


Figura 21.1 Entrada/salida vía un solo bus común.

propagación de señal de viaje redondo de 2 ns se le deben sumar varios retardos lógicos, tiempo de *buffer*, retardo de arbitraje de bus y la latencia de acceso a memoria. Además de los retardos adicionales debidos a distancia creciente desde el CPU, muchos dispositivos I/O también son más lentos según su naturaleza electromecánica.

En la tabla 3.3 se mencionan algunos dispositivos I/O junto con sus tasas de datos aproximadas y dominios de aplicación. Además del tipo de entrada o salida de datos, que se usaron en la tabla 3.3, como el criterio de clasificación primario, los dispositivos I/O se pueden categorizar con base en otras características de tecnología o aplicación. El medio que se usa para presentación de datos (copia impresa contra copia electrónica o blanda), medida del involucramiento humano (manual contra automática) y calidad de producción (bosquejo preliminar contra calidad de publicación o fotografía) son algunas de las características relevantes.

Los modernos dispositivos I/O son muy inteligentes y con frecuencia contienen su propio CPU (usualmente un microcontrolador o procesador de propósito general de bajo perfil), memoria y capacidades de comunicación, que cada vez más incluyen conectividad de red. Las interfases con tales dispositivos, que pueden tener muchos megabytes de memoria para *buffer*, son diferentes del control de los primeros dispositivos I/O, que tenían que alimentarse con información, o dirigirse para enviar datos, un byte a la vez.

Los dispositivos se comunican con el CPU y la memoria mediante un controlador I/O conectados a un bus compartido. Los sistemas simples de bajo perfil pueden usar el mismo bus para I/O que se usa para enviar datos de ida y vuelta entre el CPU y la memoria (figura 21.1). Puesto que el bus común en la figura 21.1 contiene algunas líneas de dirección para muchos accesos a memoria, es sencillo dejar que las mismas líneas seleccionen los dispositivos para operaciones I/O al asignarles una porción del espacio de dirección de memoria. Este tipo de *I/O mapeada por memoria*, y otras estrategias para control de I/O a través de programación, se cubrirán en el capítulo 22. Los sistemas más elaborados o de alto rendimiento pueden tener un bus I/O separado para garantizar que no se quita ancho de banda de las transferencias CPU-memoria (figura 21.2). Cuando existen múltiples buses en un sistema, se ponen en interfaz mutua a través de *adaptadores de bus*. En el capítulo 23 se cubrirán los buses, estándares relevantes y métodos de creación de interfases.

Los controladores I/O en las figuras 21.2 y 21.2 satisfacen las reglas siguientes:

1. Aislar el CPU y a la memoria de detalles de operación del dispositivo I/O y requisitos específicos de interfaz.
2. Facilitar la expansión en capacidades e innovación en tecnología de dispositivos I/O sin impactar el diseño y operación del CPU.
3. Gestionar las (potencialmente amplias) incompatibilidades de rapidez entre procesador/memoria en un lado y los dispositivos I/O en el otro, a través de *buffering*.
4. Convertir los formatos de datos o codificaciones y forzar los protocolos de transferencia de datos.

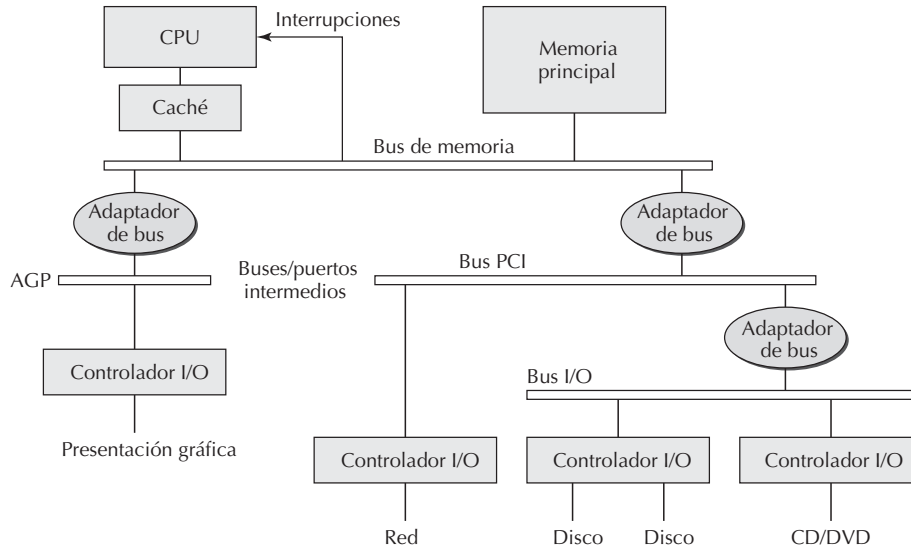


Figura 21.2 Entrada/salida mediante buses I/O intermedio y dedicado (se explicarán en el capítulo 23); el AGP (puerto gráfico acelerado) es un puerto compartido, no un bus.

Los controladores I/O son computadoras por derecho propio. Cuando se activan, corren programas especiales para guiarlos a través de las operaciones requeridas de transferencia de datos y protocolos asociados.

Por esta razón introducen a las operaciones I/O latencias no triviales que se deben considerar en la determinación del rendimiento I/O. Según los tipos de sistema y dispositivo, no son raras latencias de controlador I/O de unos cuantos milisegundos.

21.2 Teclado y ratón

Un *teclado* consiste de un arreglo de teclas que se usa para ingresar información a la computadora u otro dispositivo digital. Los teclados más pequeños, como los que se encuentran en los teléfonos, o en una área separada en muchas computadoras de escritorio, se les conoce como *teclado numérico* (*keypad*). Casi todos los teclados alfanuméricos siguen la plantilla QWERTY, nombre derivado de las etiquetas de las primeras teclas en la hilera superior de letras en el teclado de máquina de escribir estándar. Con los años se han hecho serios intentos por introducir otras plantillas de teclas que harían más fácil alcanzar letras del alfabeto de uso más frecuente, y aumentar la rapidez de entrada de datos. Se dice que la plantilla QWERTY se eligió para reducir deliberadamente la rapidez de tecleo con intención de evitar el atasco de los martillos mecánicos de las primeras máquinas de escribir. Desafortunadamente, la familiaridad con la plantilla QWERTY y la gran experiencia en su uso han evitado la adopción de estas plantillas más sensibles. Sin embargo, se ha progresado algo en el área de colocaciones de teclas no convencionales para dar al usuario comodidad durante el tecleo. Los teclados que siguen tales diseños se denominan *teclados ergonómicos*.

La figura 21.3 muestra dos diseños específicos para una tecla que es capaz de cerrar un circuito eléctrico al presionar el casquete de la tecla unida a un ensamble de émbolo o a una membrana montada sobre pistones poco profundos. Existen muchos otros diseños en uso. Los interruptores de membrana son baratos, son de uso común en las aplicaciones de sistemas incrustados, como el panel de control

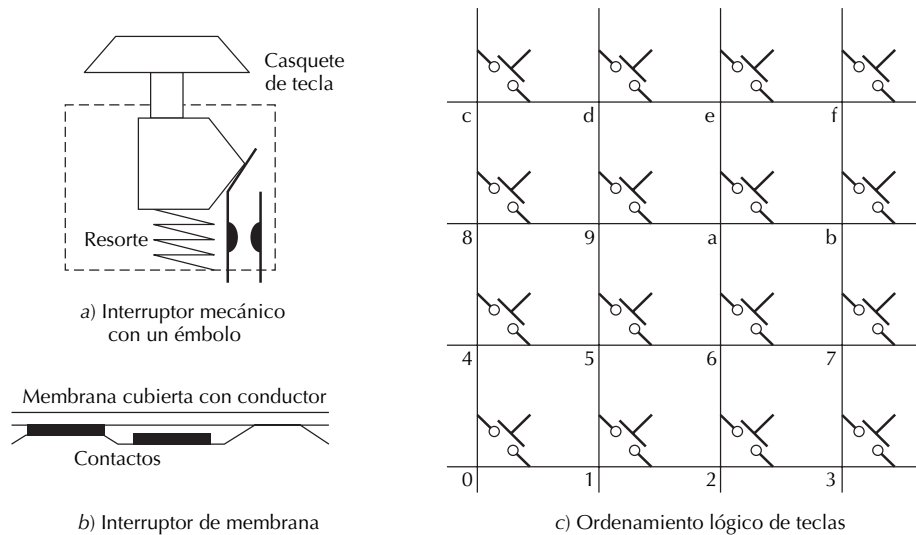


Figura 21.3 Dos diseños de interruptores mecánicos y la plantilla lógica de un teclado numérico hexa.

de un horno de microondas. Los interruptores mecánicos, como el que se muestra en la figura 21.3a, se usan en teclados de escritorio estándar en vista de la retroalimentación táctil y su construcción más fuerte. Puesto que los contactos se pueden ensuciar con el paso del tiempo, otros tipos de interruptores se apoyan en la fuerza inducida forma magnética en lugar de mecánicamente para cerrar un par de contactos. De esta forma, los contactos se pueden colocar en un coto sellado para aumentar la confiabilidad. Sin importar qué tipo de contacto se use, el circuito se puede reabrir y volver a cerrar muchas veces con cada presión de tecla. Este efecto, conocido como *rebote de contacto*, se puede evitar al no usar la señal de la tecla directamente sino dejarla que active un *flip-flop*, cuya salida se vuelve alta con una presión de tecla y permanece así sin importar la longitud o severidad del rebote de contacto. También es posible lidiar con el efecto del rebote de contacto con diseño adecuado del software que maneja adquisición de datos del teclado.

Sin importar la plantilla física, las teclas de un teclado o teclado numérico con frecuencias se ordenan en un arreglo cuadrado o rectangular 2D desde un punto de vista lógico. Cada tecla está en la intersección de un circuito de hilera y de columna. Presionar una tecla provoca cambios eléctricos en los circuitos de hilera y columna asociados con la tecla, ello permite la identificación de la tecla que se presionó (figura 21.3c). Entonces un codificador convierte las identidades de hilera y columna en código de símbolo único (usualmente ASCII), o secuencia de códigos, para que se transmitan a la computadora. La plantilla física del teclado y el ordenamiento de las teclas no son relevantes para los procesos de detección y codificación.

Con el propósito obtener la salida de cuatro bits que representa la tecla presionada en el teclado numérico hexa de la figura 21.3c, las señales de hilera se postulan a su vez, acaso con el uso de un contador de anillo de cuatro bits que sigue la secuencia de conteo 0001, 0010, 0100 y 1000. Conforme una señal se postula en una hilera particular, las señales de columna se observan. La codificación de dos bits del número de hilera unido a la codificación de dos bits del número de columna proporciona la salida de cuatro bits. Un teclado de 64 teclas puede estar ordenado lógicamente en un arreglo 8×8 (aun cuando el ordenamiento físico de las teclas pueda no ser cuadrado). Entonces se produce el código de salida de seis bits en forma similar al procedimiento para el teclado hexa. Si se desea la representación ASCII del símbolo, se puede usar una tabla de consulta con 64 entradas.

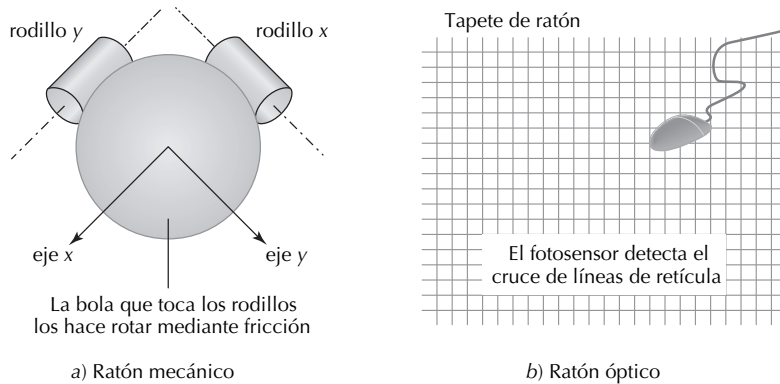


Figura 21.4 Ratones mecánico y óptico simple.

Además de un teclado, la mayoría de las computadoras de escritorio y laptop tiene dispositivos apuntadores que permitirán al usuario elegir opciones de menús, seleccionar texto y objetos gráficos para edición, y realizar una diversidad de otras operaciones. De hecho, algunos dispositivos sin teclado se apoyan exclusivamente en el señalamiento para funciones de control y captura de datos. El dispositivo apuntador usado más comúnmente es el *ratón*, llamado así por su forma en algunos de los primeros diseños y el alambre con forma de cola que lo conecta a la computadora. Los ratones modernos vienen en varias formas, muchos de ellos usan enlaces inalámbricos con la computadora.

En un ratón mecánico, dos contadores se asocian con los rodillos *x* y *y* (figura 21.4a). Levantar el ratón establece los contadores a cero. Cuando el ratón se arrastra, los contadores aumentan o disminuyen de acuerdo con la dirección de movimiento. La computadora usa los valores del contador para determinar la dirección y extensión del movimiento que se aplicará al cursor. Los ratones ópticos son más precisos y menos proclives a fallo debido a la reunión de polvo e hilachos en sus partes. Los ratones ópticos más simples detectan líneas de retícula en un tapete de ratón especial (figura 21.4b). Las versiones nuevas, más avanzadas, usan pequeñas cámaras digitales que les permiten detectar movimiento sobre cualquier superficie.

Con frecuencia, un *touchpad* (almohadilla táctil) sustituye un ratón para aplicaciones en las laptop. La ubicación de un dedo que toca el tapete se percibe (a través de circuitos de hilera y columna) y el movimiento del cursor se determina según cuán lejos, rápido y en qué dirección se movió el dedo. Una *pantalla táctil* es similar, pero, en lugar de un tapete separado, la superficie de la pantalla de presentación se usa para apuntar. Otros dispositivos apuntadores incluyen la *trackball* (esfera móvil, que en esencia representa un ratón mecánico boca abajo cuya gran bola se manipula a mano) y el *joystick* (palanca de control) que encuentra aplicaciones en muchos juegos de computadora, así como en escenarios de control industrial. Algunas computadoras laptop usan un pequeño *joystick* incrustado en el teclado para apuntar.

En la actualidad los teclados y dispositivos apuntadores son muy inteligentes y con frecuencia tienen procesadores dedicados (microcontroladores) interconstruidos.

■ 21.3 Unidades de presentación visual

La presentación visual de símbolos e imágenes es el método de salida primario para la mayoría de las computadoras. Las opciones disponibles varían desde pequeñas pantallas monocromas (de los tipos usados en los teléfonos celulares y calculadoras baratos) hasta los grandes dispositivos de presentación de alta resolución, y más bien costosos, dirigidos al uso de artistas gráficos y publicistas profesionales. Hasta hace poco el *tubo de rayos catódicos* (CRT) era el tipo principal de dispositivo de presentación

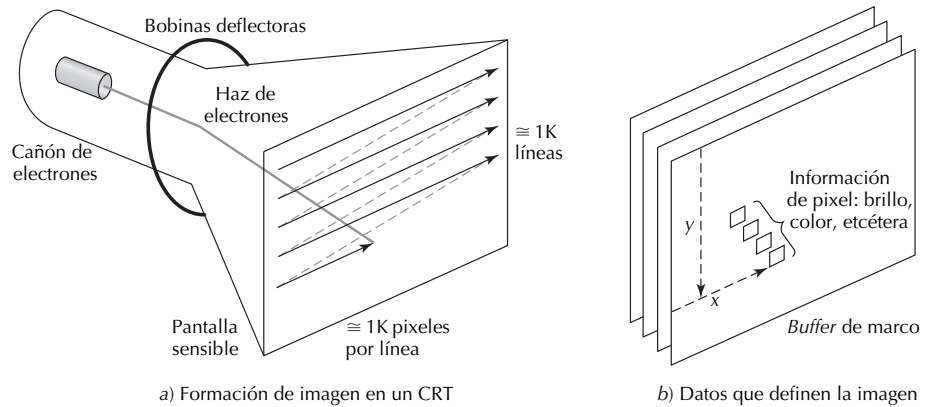


Figura 21.5 Unidad de presentación CRT y almacenamiento de imagen en *buffer* de marco.

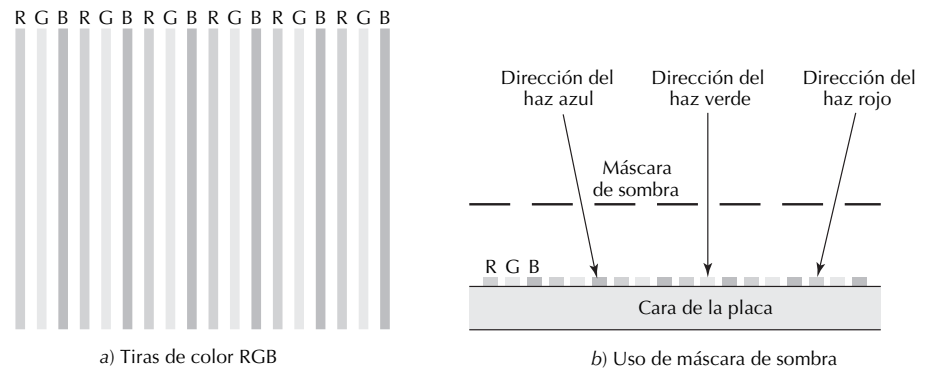


Figura 21.6 Esquema de color RGB de las modernas pantallas CRT.

visual para las computadoras de escritorio, las pantallas de panel plano se reservaban para su uso en laptop y otros dispositivos digitales portátiles. Es predecible que la mayoría de los voluminosos, pesados y grandes consumidores de electricidad CRT gradualmente serán sustituidos por las unidades de pantalla de panel plano, cuyos costos son accesibles y cuyas pequeñas huellas y menor generación de calor son ventajas importantes para el hogar y la oficina.

Las pantallas CRT funcionan mediante un haz de electrones que barre la superficie de una pantalla sensible, y crea un píxel oscuro o claro conforme pasa cada punto (figura 21.5a). Los primeros CRT eran *monocromáticos*. En tales CRT, el haz de electrones golpea una capa de fósforo en la parte posterior del vidrio de la pantalla. Esta capa de fósforo emite luz porque es golpeada por una corriente de electrones que viaja a muy alta rapidez, y la intensidad del haz de electrones que pasa un punto específico determina el nivel de brillantez. Para proteger la capa de fósforo del bombardeo directo de los electrones, detrás de ella se coloca una capa de aluminio; ésta actúa como contacto eléctrico. La tecnología del CRT de color pasó por varias etapas. Los primeros intentos usaron un CRT monocromático que se veía mediante la rotación de filtros de color mientras se desplegaban a la vez los píxeles asociados con varios colores. Otros diseños incluyeron la sustitución de los filtros mecánicos con otros controlados electrónicamente y el uso de múltiples capas de fósforo, cada uno emitiendo luz de color diferente.

Los CRT modernos en uso común actualmente se basan en el método tricolor de “máscara de sombra” introducido por los televisores RCA en la década de 1950. Los detalles de diseño varían, pero

es representativo el esquema que se usa en los tubos Trinitron de Sony. Como se muestra en la figura 21.6, estrechas tiras de fósforo que producen tres colores de luz diferentes y que se colocan en la parte posterior de la cara de vidrio del tubo. Los tres colores son rojo, verde y azul, que propicia el nombre “RGB” (por sus siglas en inglés) al esquema resultante. Tres haces de electrones separados exploran la superficie de la pantalla, cada uno llega en un ángulo ligeramente diferente. Una máscara de sombra, que representa una placa metálica con aberturas u hoyos, fuerza a cada haz a impactar sólo tiras de fósforo de un color particular. La separación de dos tiras consecutivas del mismo color, conocida como *densidad de presentación (display pitch)*, dicta la resolución de la imagen resultante.

Los tres haces de electrones se controlan con base en una representación de la imagen que se desea presentar. Todo pixel está asociada con cuatro (imágenes simples de 16 colores) a 32 bits de datos en un *buffer de marco*. Si la resolución de la pantalla es de $1\text{ K} \times 1\text{ K}$ pixeles, y cada pixel tiene 64K colores posibles, entonces el *buffer* de marco necesita 2 MB de espacio. Este espacio se puede proporcionar en una *memoria de video* dedicada o como parte del espacio de dirección de memoria principal (*memoria compartida*). Con frecuencia, las memorias de video dedicadas son de puerto dual para permitir el acceso simultáneo mediante el CPU para modificar la imagen o por el controlador de pantalla para presentarla. A tales memorias de video a veces se les conoce como VRAM. La figura 21.5 muestra el movimiento de barrido del haz de electrones, conforme cubre toda la superficie de la pantalla sensible 30-75 veces por segundo (conocida como *velocidad de regeneración, refresh rate*) y un *buffer* de marco que almacena cuatro bits de datos por pixel. En la práctica, hasta 32 bits de datos se necesitan por pixel:

32 bits, ocho para cada R, G, B, A (“color verdadero”).

16 bits, cinco para cada R y B, seis para G (“color alto”).

Ocho bits o 256 diferentes colores en formato VGA.

En las descripciones mencionadas, “A” significa “alfa” (un cuarto componente que se usa para controlar la mezcla de colores) y VGA representa un antiguo arreglo gráfico de video que todavía se soporta por compatibilidad.

Ejemplo 21.1: Rendimiento total de video Considere una unidad de presentación visual con una resolución de 1024×768 pixeles y una velocidad de regeneración de 70 Hz. Calcule el rendimiento total requerido por la memoria de video para soportar este despliegue, si supone ocho, 16 o 32 bits de datos por pixel.

Solución: El número de pixeles a los que se accede por segundo es $1024 \times 768 \times 70 \cong 55\text{M}$. Esto último implica un rendimiento total de 55 MB/s, 110 MB/s o 220 MB/s, según el ancho de datos de pixel. Con un acceso de 55M pixeles por segundo, están disponibles alrededor de 18 ns para cada lectura de pixel, incluso si se ignora el tiempo desperdiciado debido a márgenes izquierdo y derecho y el tiempo de retorno del haz. En consecuencia, si se deben lograr las tasas de datos deseadas con VRAM típicas, en cada acceso se deben leer múltiples pixeles.

Físicamente, las pantallas CRT son voluminosas, requieren alto consumo de energía y generan gran cantidad de calor. Además, la imagen presentada por un CRT sufre de distorsión así como de foco y color no uniformes. Por estas razones, las pantallas de cristal líquido en panel plano (LCD) son populares, en especial a la luz de su continua mejora en la calidad de imagen y su costo económico. La figura 21.7a muestra una pantalla de *matriz pasiva*. Cada punto de intersección entre una hilera y una columna representa una pequeña persiana óptica que se controla mediante la diferencia entre los

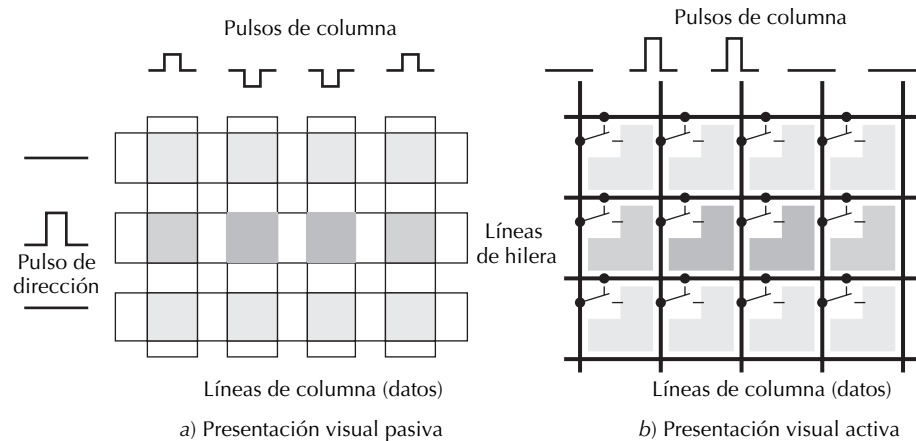


Figura 21.7 Presentaciones visuales LCD pasiva y activa.

voltajes de hilera y columna. Puesto que el voltaje de hilera se proporciona a toda celda en una hilera y las celdas además reciben voltaje cruzado de otras hileras, el contraste y la resolución tienden a ser bajos. Las *matrices activas* son similares, excepto que en cada intersección se planta un transistor de película delgada. Un transistor que se enciende permite que las celdas de cristal líquido asociadas se carguen a los voltajes en las líneas de columna. De esta forma, una imagen que se “escribe” hilera por hilera se conserva hasta el siguiente ciclo de regeneración.

Otras tecnologías de pantalla de panel plano incluyen las que se basan en diodos emisores de luz (LED, por sus siglas en inglés) y el fenómeno de plasma. Las pantallas LED consisten de arreglos de LED de uno o más colores, y cada LED se “direcciona” a través de un índice de hilera y otro de columna. Los LED electrónicos son adecuados para presentaciones que se deben ver en exteriores, requieren enorme grado de brillantez. Un desarrollo reciente en las pantallas de LED es el surgimiento de los LED orgánicos (OLED, por sus siglas en inglés), que requieren menos energía que los LED ordinarios y ofrecen una variedad de colores, incluido el blanco. La operación de las pantallas de plasma es similar a la de una lámpara de neón. Explotan la propiedad de ciertas mezclas de gases que se descomponen en plasma cuando se sujetan a un campo eléctrico muy intenso. El plasma conduce la electricidad y convierte una parte de la energía eléctrica en luz visible. En la actualidad, las pantallas de plasma son muy caras; por tanto, tienen aplicaciones limitadas.

Muchas computadoras también ofrecen salida compatible con el formato de presentación en televisión, ello permite que cualquier TV actúe como unidad de presentación. La misma señal compatible con TV se puede proporcionar a un *proyector de video* que produce una réplica de la imagen en el CRT o unidad de pantalla de panel plano de la computadora en una pantalla del tamaño de una pared. Estas tecnologías de presentación son útiles para presentaciones basadas en computadora en todas partes, desde pequeñas salas de juntas o salones de clase hasta grandes auditorios.

■ 21.4 Dispositivos de entrada/salida de copia impresa

A pesar de las numerosas predicciones de que las oficinas sin papel, o acaso una sociedad sin papel, harían obsoletos los dispositivos I/O de copia impresa, aún existe la necesidad de tales equipos. Los documentos en papel, lejos de caer en desuso, han proliferado con el uso creciente de computadoras en los negocios y escenarios domésticos.

La entrada de copia impresa se acepta a través de *escáneres* (*digitalizadores*). Un digitalizador funciona como una unidad de pantalla CRT, pero en dirección contraria. Mientras que una pantalla CRT

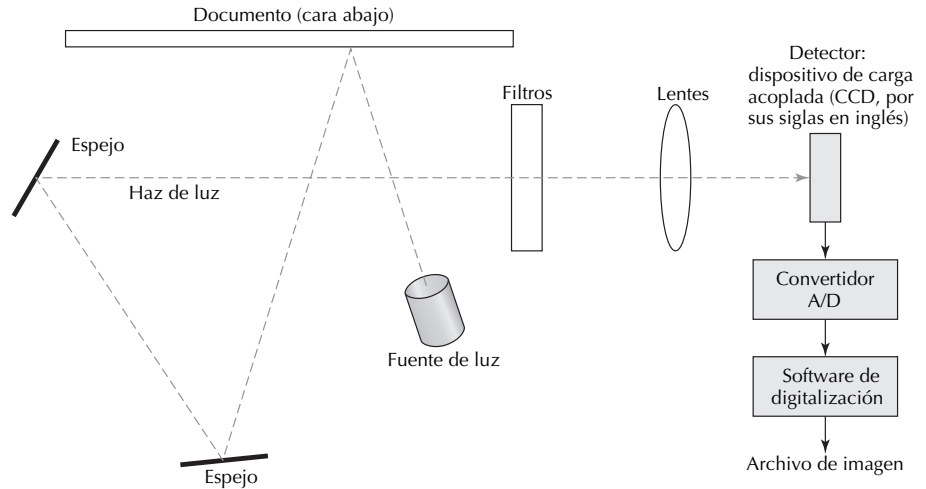


Figura 21.8 Mecanismo de digitalización para entrada de copia impresa.

convierte señales electrónicas en niveles de color y brillo con un movimiento de barrido, un escáner percibe el color y la intensidad y los convierte en señales electrónicas conforme recorre una imagen línea por línea y punto a punto dentro de cada línea (figura 21.8). Esta información se almacena en memoria en uno de muchos formatos estándar. Los escáneres se caracterizan por resolución espacial, expresada en número de píxeles o puntos por pulgada (por ejemplo, 1 200 dpi) y fidelidad de color (número de colores). En relación con el tipo de entrada de documento al escáner, tales dispositivos se dividen en los tipos *alimentador de hoja*, *cama plana*, *de cabeza* y *portátiles*. Los escáneres de cama plana tienen la ventaja de no provocar daño al documento original debido a doblado y permiten la digitalización de un libro y páginas de periódico. Los escáneres de cabeza y portátiles pueden digitalizar documentos grandes sin ser voluminosos o costosos. Los escáneres baratos están haciendo obsoletas las máquinas de fax.

Para el caso del texto digitalizado, la imagen se puede convertir en forma simbólica a través de software de *reconocimiento óptico de caracteres* (OCR, por sus siglas en inglés) y se almacena como archivo de texto. Este tipo de conversión de imagen a texto mediante digitalización y OCR se usa comúnmente para poner en línea libros antiguos y otros documentos. El reconocimiento de la escritura manual también se ha vuelto cada vez más importante conforme proliferan las aplicaciones de los *asistentes digitales personales* (PDA, por sus siglas en inglés) y las *computadoras portátiles* sin teclado.

El desarrollo de impresoras modernas es una de las historias de éxito que inspiran temor en informática. Las primeras impresoras para computadora funcionaban en forma parecida a las antiguas máquinas de escribir mecánicas; usaban dispositivos de formación de caracteres que golpeaban sobre una banda de tela o plástico mediante un mecanismo parecido a un martillo para imprimir los caracteres sobre papel uno tras otro (*impresoras de caracteres*). Para aumentar la rapidez de tales impresoras, se desarrollaron mecanismos especiales que permitían la impresión de líneas completas a la vez, ello condujo a una amplia variedad de *impresoras de línea*. Gradualmente, las impresoras de línea evolucionaron de las ruidosas y voluminosas máquinas (con tamaño de refrigerador) a unidades más pequeñas y silenciosas. Pronto hubo conciencia de que la formación de caracteres mediante la selección de un subconjunto de puntos en una matriz 2D de puntos (figura 21.9) conduciría a mayor flexibilidad en los conjuntos de caracteres arbitrarios de soporte, así como en la formación de imágenes. Por tanto, las *impresoras de matriz de punto* gradualmente sustituyeron a muchas tecnologías de impresión del tipo impacto.

Las impresoras modernas básicamente imprimen una gran imagen en matriz de punto de la página que está compuesta de archivos de texto o gráfico, mediante *postscript* u otros formatos intermedios de

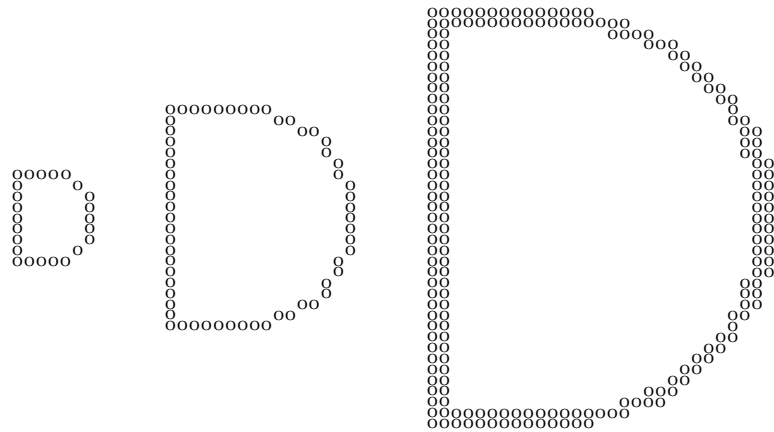


Figura 21.9 Formación de la letra “D” mediante matrices de punto de varios tamaños.

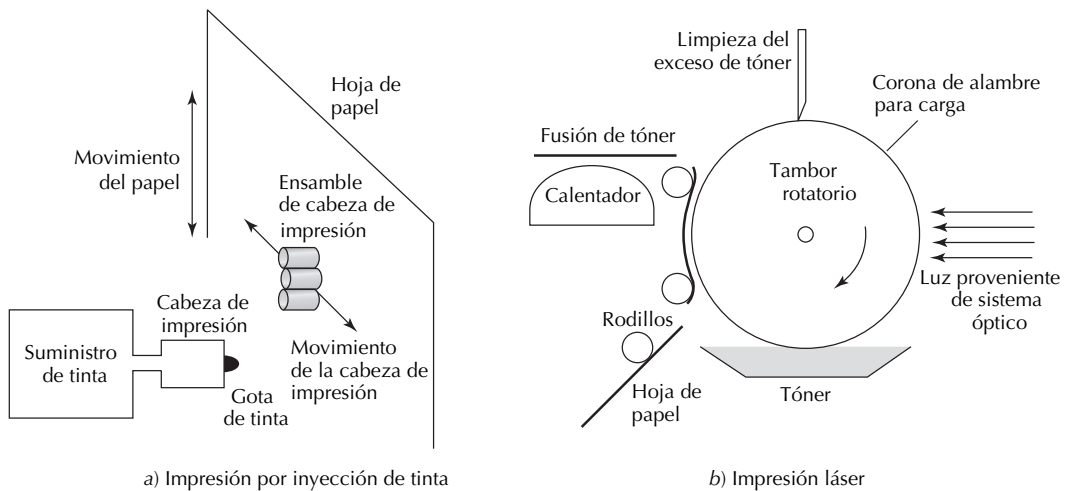


Figura 21.10 Impresoras de inyección de tinta y láser.

archivo de impresora. Cada punto en la imagen de la página se caracteriza por intensidad y color. Las *impresoras de inyección de tinta* (figura 21.10a), relativamente lentas y baratas, imprimen los puntos uno o pocos a la vez. Las gotas de tinta se expulsan desde la cabeza de impresión mediante varios mecanismos como calentamiento (que conduce a la expansión de una burbuja de aire dentro de la cabeza) o generación de ondas de choque a través de un transductor de cristal piezoeléctrico al lado del depósito. Las *impresoras láser* más grandes y rápidas (figura 21.10b) forman una copia de la imagen a imprimir en la forma de patrones de carga eléctrica en un tambor rotatorio e imprimen toda la página a alta rapidez.

Al igual que un escáner, una impresora de matriz de punto se caracteriza por su resolución espacial, expresada en número de píxeles o puntos por pulgada (por ejemplo, 1 200 dpi), y fidelidad de color (número de colores). El rendimiento total de impresión, otra característica clave de rendimiento para las impresoras de computadora, varía de unas cuantas a muchos centenares de páginas por minuto (ppm), en ocasiones varían incluso para la misma impresora dependiendo de la resolución y requisitos de color. Muchas impresoras antiguas requerían papel especial cubierto con químicos o empacado en rollos para simplificar el mecanismo de alimentación de papel. Las impresoras modernas usan papel ordinario, se denominan *impresoras de papel normal*.

Las impresoras para blanco y negro depositan gotas de tinta o funden partículas de tóner en el papel de acuerdo con los requisitos del documento. Los niveles de gris se crean al dejar que se muestre parte del blanco del papel subyacente. Las impresoras de color funcionan de modo similar a los CRT de color en que crean varios colores a partir de tres colores. Sin embargo, los tres colores que se usan en las impresoras son cian (azul-verde), magenta y amarillo (*yellow*), que en conjunto forman el esquema de color CMY (por sus siglas en inglés), es diferente del RGB de los CRT. Las razones para la diferencia tienen que ver con la forma en que el ojo humano percibe el color. El esquema de color CMY representa la ausencia de los colores RGB (cian es la ausencia de rojo, etc.). De este modo, el cian absorbe rojo, el magenta absorbe verde y el amarillo absorbe azul. El esquema de color RGB es aditivo, ello significa que un color deseado se crea al sumar la cantidad adecuada de cada uno de los tres colores primarios. El esquema CMY es sustractivo y forma un color deseado al remover los componentes adecuados de la luz blanca. La mezcla de estos tres colores en cantidades iguales absorbería los tres colores primarios y dejaría negro. Sin embargo, el blanco así producido es más bien insatisfactorio, especialmente en vista de la extrema sensibilidad del ojo humano ante cualquier color corrido en negro. Por esta razón, la mayoría de las impresoras de color usan el esquema CMYK, donde K representa al negro.

La impresión a color es mucho más complicada que la pantalla a color. Las razones incluyen la dificultad de controlar exactamente el tamaño y alineación de los puntos de los diversos colores y la posibilidad de que los colores se corran si se colocan muy juntos. Mayores problemas surgen de la resolución reducida cuando se deben soportar diferentes niveles de intensidad (escalas de grises en negro y blanco). Por ejemplo, una forma de crear la ilusión de cinco niveles de gris (0, 25, 50, 75 y 100%) es dividir el área de impresión en bloques de píxeles de 2×2 , y colocar 0-4 píxeles negros en un bloque para crear los cinco niveles. Sin embargo, esto último reduce la resolución por un factor de 2 en cada dirección (un global cuadruplicado).

Aunque las impresoras modernas pueden producir salida de copia impresa de alta calidad, también están disponibles dispositivos de salida de copia impresa especializada para requerimientos particulares. Los ejemplos incluyen *plotters* (trazadores), que se usan para producir dibujos técnicos y arquitectónicos, usualmente en papeles muy grandes, y las *impresoras fotográficas*, que difieren de las impresoras normales sólo en la calidad de sus mecanismos de impresión y los tipos de papel que aceptan.

El uso de máquinas de oficina que combinan escáner/impresora es cada vez más común. Tales máquinas pueden ofrecer capacidades de transmisión de fax y fotocopiado con poco hardware adicional. Las máquinas de fax digitalizan documentos en archivos de imagen antes de la transmisión, de modo que, cuando hay presente capacidad de digitalización, el resto es sencillo. De hecho, los documentos cada vez más se envían mediante escaneo seguido por transmisión como adjunto (*attachment*) a un correo electrónico, en lugar de máquinas de fax. El copiado es, en esencia, digitalización seguido por impresión.

Ejemplo 21.2: Rendimiento total de datos de una copiadora digital Si supone una resolución de 1200 dpi, área de copiado de 8.5 pulgadas \times 11 pulgadas y color de 32 bits en una copiadora digital compuesta de un escáner seguido por una impresora láser, derive el rendimiento total de datos necesarios para soportar un rendimiento total de copiado de 20 ppm (páginas por minuto).

Solución: La tasa de datos para imprimir un tercio de una página por segundo es $8\frac{1}{2} \times 11 \times 1200^2 \times \frac{4}{3} \cong 180$ MB/s. Si supone que los datos provenientes del escáner se almacenan en una memoria y luego se recuperan para impresión, la tasa de datos a soportar por la memoria es de 360 MB/s. Esto está bien adentro de las capacidades de las modernas DRAM. Sin embargo, es posible reducir sustancialmente esta tasa al aprovechar el espacio en blanco de la mayoría de los documentos (compresión de datos).

■ 21.5 Otros dispositivos de entrada/salida

Además de las memorias secundaria y terciaria que constituyen los dispositivos I/O usados comúnmente para almacenamiento estable y archivado de datos (capítulo 19), están disponibles muchos otros tipos de unidades de entrada y/o salida. En esta sección se revisan las más importantes de dichas opciones para completar el estudio acerca de los dispositivos I/O.

Una cámara digital fija o de video captura imágenes para ingresar a una computadora en forma muy similar a un escáner. La luz entrante proveniente del exterior de la cámara se convierte en píxeles y se almacena en memoria *flash* o algún otro tipo de unidad de memoria no volátil. Las cámaras digitales fijas se caracterizan por su resolución en términos del número de píxeles en cada imagen capturada. Por ejemplo, una cámara de un megapixel puede tener una resolución de 1280×960 . Las cámaras con resoluciones de cinco o más megapíxeles son muy costosas y usualmente no se necesitan en aplicaciones sofisticadas, pues una cámara de dos megapíxeles puede entregar impresiones de calidad fotográfica de 20×25 cm en impresoras de inyección de tinta. Las cámaras digitales pueden usar zoom óptico tradicional para acercar los objetos; también pueden tener zoom digital que usa algoritmos de software para acercar una parte particular de la imagen digital, pero en el proceso se reduce la calidad de la imagen. Algunas cámaras digitales fijas pueden tomar películas cortas de resolución más bien baja. Las videocámaras digitales son capaces de capturar videos de gran calidad, mientras que las webcams se usan para capturar imágenes en movimiento con el propósito de dar seguimiento, videoconferencias y aplicaciones similares en las que la calidad y suavidad del movimiento de la imagen no son cruciales. Las fotografías y películas se almacenan en las computadoras en una diversidad de formatos estándar, usualmente en forma comprimida para reducir los requisitos de almacenamiento. Los ejemplos más comunes incluyen JPEG (por sus siglas en inglés, *Joint Photographic Experts Group* = Grupo conjunto de expertos fotográficos) y GIF (por sus siglas en inglés, *Graphic Interchange Format* = Formato de intercambio gráfico), para imágenes, y MPEG y Quick Time para películas.

Imágenes tridimensionales se pueden capturar para su proceso por computadora mediante escáneres de volumen. Un método consiste en proyectar una línea láser sobre el objeto 3D para digitalizarlo y capturar la imagen de la línea donde interseca el objeto mediante cámaras de alta resolución. A partir de imágenes capturadas y la información acerca de la posición y orientación de la cabeza que digitaliza, se calculan las coordenadas de superficie del objeto.

La entrada de audio se captura mediante micrófonos. La mayoría de las computadoras de escritorio y laptop vienen con micrófono, o un par estéreo, y contiene una tarjeta sonora que puede capturar sonidos mediante un puerto de micrófono. Para almacenar en una computadora, el sonido se digitaliza al tomar muestras de la forma ondulatoria a intervalos regulares. La fidelidad del sonido se mejora al aumentar la tasa de muestreo y al usar más bits para representar cada muestra (por decir, 24 bits para resultados profesionales, en lugar de 16 bits). En virtud de que estas provisiones conducen a aumento en los requisitos de almacenamiento, usualmente el sonido se comprime antes de almacenarlo. Los ejemplos de formatos estándar para audio comprimido incluyen MP3 (abreviatura para MPEG-1, capa 3), Real Audio, Shockwave Audio y el formato WAV de Microsoft Windows. MP3 comprime los archivos de audio hasta casi 1 MB por minuto y conduce a audio con calidad de CD porque en el curso de la compresión remueve sólo componentes de sonido que superan el rango auditivo humano. Además de esos formatos, se pueden usar los de película MPEG y Quick Time para almacenar sonido al ignorar el componente de video.

Ahora es frecuente el uso de sensores para proporcionar información acerca del ambiente y otras condiciones de interés para las computadoras. Por ejemplo, existen tableros de sensores en un automóvil y muchos miles de aquéllos en una planta industrial moderna. A continuación se presenta una lista parcial de sensores usados comúnmente y sus aplicaciones.

- Las fotoceldas constituyen los *sensores de luz* más simples para controlar luces nocturnas, semáforos, sistemas de seguridad, cámaras y juguetes. Una fotocelda incorpora una resistencia variable que cambia con la luz (desde muchos miles de ohms en la oscuridad hasta casi un kilohm en luz brillante), ello facilita detectar la cantidad de luz mediante un convertidor analógico-digital.
- Los sensores de temperatura son de dos tipos. Un detector de contacto mide su propia temperatura y deduce un objeto con el que está en contacto, si supone equilibrio térmico. La medición se basa en la variación de las propiedades del material (como resistencia eléctrica) con la temperatura. Los detectores de no contacto miden la radiación infrarroja u óptica que reciben de un objeto.
- Los detectores de presión convierten la deformación o alargamiento en materiales a cambios en algunas propiedades eléctricas mensurables. Por ejemplo, el alambre en zigzag incrustado en un sustrato plástico en un *extensímetro* debido a deformación, con lo que su resistencia cambia ligeramente. Los detectores de presión microelectromecánica ofrecen mayor sensibilidad y precisión. Los detectores de presión se usan en tuberías, control de motores, alas de aviones etcétera.

Las innovaciones en el diseño de sensores son de inmenso interés como resultado de la creciente demanda en el control incrustado, los sistemas de seguridad y aplicaciones militares. Las nuevas tecnologías que se siguen incluyen sensores microelectromecánicos (MEM), particularmente para presión, y biosensores, que incorporan componentes biológicos, ya sea en los mecanismos usados para detectar o en el fenómeno detectado. Los sensores MEM ofrecen mayor sensibilidad y precisión, así como tamaño pequeño. Las ventajas de los biosensores incluyen bajo costo, mayor sensibilidad y economía de energía. Se usan en control de la contaminación, análisis bacterial, diagnóstico médico y minado, entre otras aplicaciones.

La salida de imagen mediante computadoras se produce al “renderizar” varios tipos de dispositivos de presentación visual (sección 21.3) o imprimir/graficar en papel (sección 21.4). De manera adicional, las imágenes se pueden transferir directamente a microfilme para archivar o proyectar en una pantalla durante presentaciones audiovisuales. Lo último se hace usualmente al conectar un videoproector al puerto de salida de pantalla de una computadora de escritorio o laptop o a un puerto especial que ofrece salida para formato de TV. Las salidas gráficas más exóticas incluyen imágenes estereográficas para aplicaciones de realidad virtual e imágenes holográficas.

La salida de audio se produce a partir de archivos de audio, y la tarjeta de sonido convierte archivos almacenados posiblemente comprimidos en ondas eléctricas adecuadas para enviar a una o más bocinas. Además de los formatos de archivo de audio ya mencionados en conexión con la entrada de audio, la salida de sonido se puede producir a partir de archivos MIDI (por sus siglas en inglés, *Musical Instrument Digital Interface*, interfaz digital de instrumento musical) mucho más compactos. Los archivos MIDI no contienen audio grabado sino instrucciones con el fin de que la tarjeta de sonido produzca notas particulares para ciertos instrumentos musicales. Por esta razón, los archivos MIDI usualmente son 100 o más veces más pequeños que archivos MP3 comparables, pero tienen mucho menos fidelidad. Los sintetizadores de habla también constituye un área de gran interés, pues permite una interfaz de usuario más natural en muchos contextos. La síntesis de habla se puede realizar mediante la unión de fragmentos pregrabados o con conversión algorítmica texto a habla.

Ahora es común que las computadoras se usen para control directo de una diversidad de dispositivos mecánicos. Esto último se hace mediante *actuadores* que convierten señales eléctricas a fuerza y movimiento. Los *motores de velocidad gradual* (también llamados *motores de paso*) son los componentes más usados. Un motor de velocidad gradual es capaz de pequeñas rotaciones al recibir uno o más pulsos de control. Entonces una rotación se puede convertir a movimiento lineal, si se desea, o usarse directamente para mover cámaras, brazos robóticos y muchos otros tipos de dispositivos. La figura 21.11 muestra las funciones de un motor de velocidad gradual típico. El rotor consta de imanes

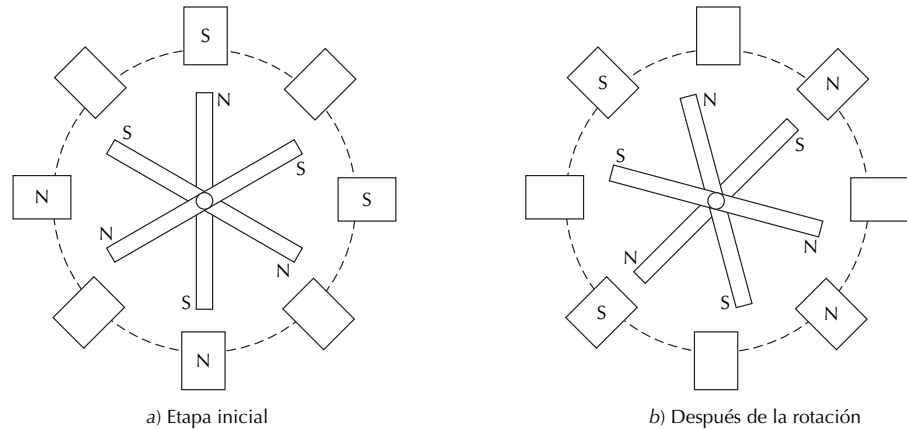


Figura 21.11 Principios de operación del motor de velocidad gradual.

separados 60° . El estator se divide en ocho secciones con 45° de espaciado. Suponga que la mitad de los segmentos de estator se magnetizan como se muestra en la figura 21.11a). Ahora, si la mitad restante de los segmentos de estator se magnetizan como en la figura 21.11b) y la corriente de magnetización se remueve del primer conjunto, el rotor gira 15° (la diferencia entre el espaciado de 60° del rotor y el espaciado de 45° del estator).

Hay alguna esperanza de que, eventualmente, dispositivos sin partes mecánicas permitirán la construcción de generadores de movimiento más pequeños, más ligeros, más fuertes y de bajo consumo eléctrico. Los *polímeros electroactivos* parecidos a músculos, que se expanden y contraen en respuesta a la estimulación eléctrica, representan una tecnología útil que actualmente se persigue [BarC01].

■ 21.6 Redes de dispositivos de entrada/salida

En forma creciente, entrada y salida involucran transferencias de archivos mediante una red. Por ejemplo, enviar un documento impreso a una impresora que tiene un gran *buffer* de datos y un CPU no es muy diferente de enviar un archivo a otra computadora. Los dispositivos periféricos basados en red permiten compartir recursos inusuales o rara vez usados y también ofrecen opciones de respaldo en el evento de falla o sobrecarga de un recurso local (figura 21.12). Por ejemplo, en un escenario de oficina, un grupo de trabajo puede compartir una costosa impresora láser a color, con otra impresora ubicada en un departamento diferente, designada como respaldo en caso de descompostura. Las I/O en red también ofrecen flexibilidad en la colocación de dispositivos I/O de acuerdo con necesidades dinámicamente cambiantes y permiten la actualización o sustitución, mientras que reduce el cableado y los problemas de conectividad. Más aún, según la universalidad de los estándares de comunicación en red, como IP y Ethernet (sección 23.1), las I/O en red mejoran la compatibilidad y facilitan la interoperabilidad.

En ninguna parte los beneficios de las I/O en red son más evidentes que en el control de procesos industriales. En tales sistemas, tableros de cientos o miles de sensores, actuadores y controladores deben interactuar con una computadora central o con nodos de plataforma distribuida. El uso de una red con una topología de árbol o ciclo, en lugar de conexiones punto a punto, conduce a reducción significativa en el cableado, con la acompañante reducción en costos de operación y mantenimiento. La misma red puede usarse para intercambiar datos, diagnósticos, configuración y calibración, ello ahorra costos. Tal nivel reducido de alambrado se puede evitar si se usa red inalámbrica.

En el contexto anterior, entrada/salida se mezcla ampliamente con funciones de procesamiento y control, y las fronteras entre las diversas funciones se vuelven confusas. Un sensor con un procesador

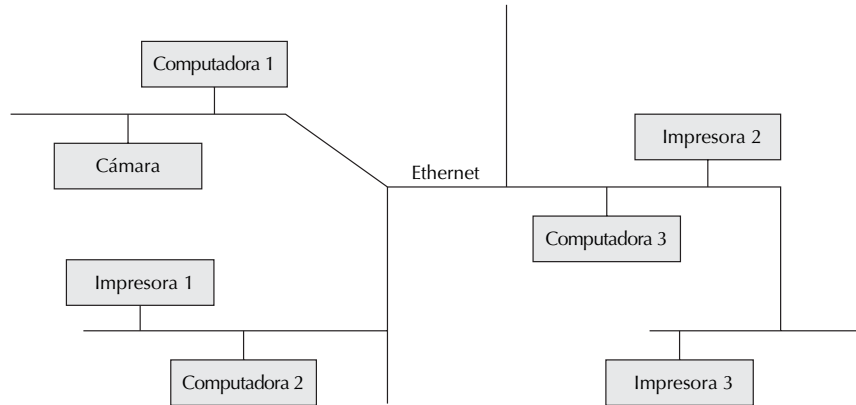


Figura 21.12 Con periféricos habilitados por red, la I/O se realiza mediante transferencia de filtros.

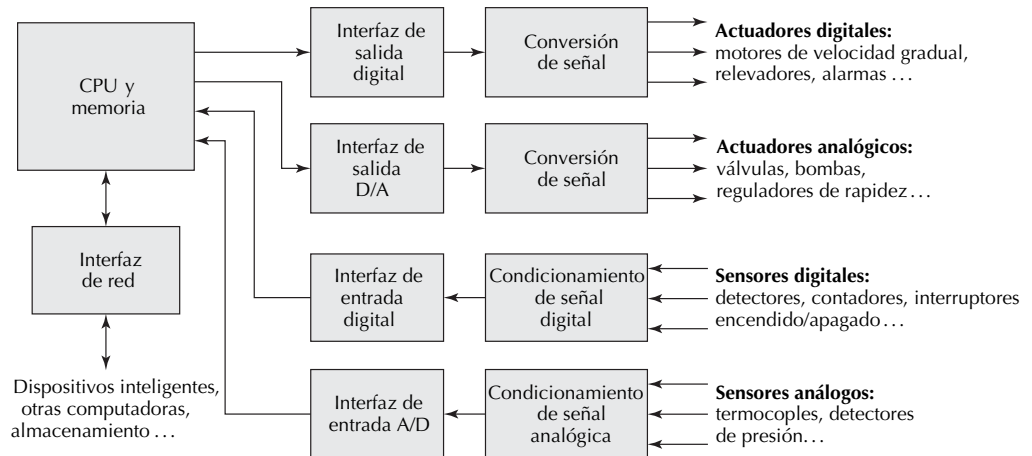


Figura 21.13 La estructura de un sistema de control de ciclo cerrado basado en computadora.

interconstruido y adaptador de red, integrado en un solo chip que use MEM y otras tecnologías, se visualiza como un dispositivo de entrada o como un nodo en un sistema de procesamiento distribuido (sección 28.4). Se trata de un dispositivo de entrada en el sentido de que su principal función y meta final es proporcionar información acerca de su entorno a un proceso de control o algoritmo. Es un nodo en el sentido de que es capaz de cooperar con otros nodos, correr autodiagnósticos, realizar autocalibración o calibración mutua, y realiza varias funciones relacionadas con recolección de datos tradicionalmente relegadas a un procesador central. Lo último incluye tareas relacionadas a operación confiable y precisa: filtrar ruido, reintento de transmisión, adaptación a cambios en el entorno, etcétera.

La figura 21.13 muestra la estructura de un sistema de control de ciclo cerrado basado en computadora con interfases especializadas para manejar sensores y actuadores de varios tipos. Con dispositivos habilitados por red, la mayoría de éstos desaparecen o se incorporan en el subsistema sensor/actuador. Todo pasa por la computadora de control vía la interfaz de red, ello simplifica su labor. El diseñador del software para tal sistema de control basado en computadora pueden enfocarse en los requisitos algorítmicos y de rendimiento relevantes, no así en las necesidades de interfaz específicas para cada tipo de dispositivo.

PROBLEMAS

21.1 Teclados y sus interruptores

- Los interruptores mecánicos que se muestran en la figura 21.3a parecen requerir gran cantidad de movimiento vertical para cerrar el contacto eléctrico. Si examina el teclado en una moderna computadora notebook ultradelgada, observará que las teclas se mueven muy poco. Investigue el diseño de tales interruptores de movimiento pequeño.
- Algunos teclados se publicitan como “a prueba de derrames”. ¿Cuáles son las implicaciones de esta propiedad sobre el diseño de interruptores y otras partes de un teclado?
- Mencione otras dos propiedades físicas de un teclado que considere importantes para un usuario.
- Mencione dos propiedades negativas de un teclado que conducirían a insatisfacción o rechazo del usuario.

21.2 Codificación de teclado

Diseñe un codificador para el teclado numérico de la figura 21.3c y suponga que la salida debe ser:

- Un dígito hexa de cuatro bits.
- Un carácter ASCII de ocho bits que represente al dígito hexa.

21.3 Codificación de teclado

- La discusión de codificación de teclado en la sección 21.2 supuso tácitamente que cuando mucho una tecla se presiona en un momento dado. La mayoría de los teclados soportan dobles presiones de teclas (por ejemplo, una tecla ordinaria más una de las teclas especiales shift, alt o control) para permitir un conjunto más amplio de símbolos. Especule acerca de cómo se logra la codificación en este caso.
- En máquinas basadas en Windows, al menos se soporta una triple presión de tecla (Alt + control + delete). ¿Puede pensar en una buena razón para incluir esta característica, que ciertamente complica el proceso de codificación?
- En las previsiones de las partes a) y b), múltiples presiones de tecla conducen a la transmisión de un símbolo del teclado a la computadora. Lo opuesto también puede ser cierto para algunos teclados: múltiples símbolos enviados como resultado de una sola presión de tecla. ¿Por qué es útil esta característica? Describa una forma de efectuar el codificador requerido.

21.4 Dispositivos apuntadores

Realice el siguiente experimento en la touchpad de $5 \times 6 \text{ cm}^2$ de una computadora laptop. Mueva el cursor a la esquina superior izquierda de la pantalla de $21 \times 28 \text{ cm}^2$. Coloque un dedo en la esquina superior izquierda del touchpad y arrástrelo rápidamente hacia la esquina inferior derecha de la almohadilla. La nueva posición del cursor está cerca de la esquina inferior derecha de la pantalla. Ahora, mueva el dedo en la dirección opuesta a lo largo de la misma ruta diagonal, pero más lentamente. Cuando el dedo llegue a la esquina superior izquierda de la touchpad, el cursor está cerca del centro de la pantalla. ¿Qué le dice este experimento acerca de cómo se procesan los datos de posición del touchpad? Justifique sus conclusiones y discuta sus implicaciones prácticas.

21.5 Rendimiento total de la memoria de video

En el ejemplo 21.1 la tasa de datos se derivó con la suposición de que todos los píxeles en el *buffer* de marco se deben actualizar en cada marco.

- Mencione tres aplicaciones en las que sólo una fracción muy pequeña de los píxeles cambie de un marco al siguiente.
- Describa una forma de enviar datos hacia el *buffer* de marco que permite actualización selectiva de los píxeles o pequeñas ventanas dentro del área de presentación.
- Discuta la cabecera debida a la transmisión selectiva de la parte b), tanto en términos de envío de bits adicionales (además de los datos de píxel reales) como de tiempo de procesamiento adicional necesario para extraer los datos de píxel.

21.6 Tablero de marcador

Algunas pantallas de tablero de marcador monocromas se construyen a partir de arreglos 2D de bombillas luminosas que se pueden encender y apagar bajo control programado.

- Describa cómo se puede controlar un tablero de este tipo de 100×250 mediante 16 señales de datos provenientes de un microcontrolador.
- Discuta los beneficios y perjuicios de dos formas de ordenar las bombillas en el tablero: la j -ésima bombilla de la hilera $2i + 1$ alineada verticalmente con la

j -ésima bombilla de la hilera $2i$, contra su alineación con el punto medio entre la j -ésima y la $(j + 1)$ -ésima bombillas en la hilera $2i$.

- c) ¿Es factible construir un tablero de color de este tipo?

21.7 Resolución ajustable en monitores

Los diseños y métodos descritos para monitores CRT y de panel plano de la sección 21.3 sugieren espaciamientos fijos de pixel. Sin embargo, en una computadora personal típica, el usuario puede fijar la resolución (número de pixeles en la pantalla) mediante un programa de utilidad de sistema. ¿Cómo se implementan realmente dichas variaciones? En otras palabras, las aberturas de la máscara de sombra en la figura 21.6b o las líneas de hilera y columna en la figura 21.7 no cambian. De este modo, ¿qué cambia cuando se varía la resolución?

21.8 Escáner

Las especificaciones para un escáner indican que tiene una resolución de 600 dpi en la dirección x y 1 200 dpi en la dirección y .

- ¿Por qué cree que las resoluciones son diferentes en las dos direcciones?
- ¿Qué tipo de procesamiento de software permitiría imprimir una imagen capturada por este escáner en una impresora de 1 200 dpi?
- Repita la parte b) para una impresora de 600 dpi.

21.9 Rendimiento total de datos en una copiadora

Un memorando típico de oficina, la página de un libro u otro documento que se copia contienen principalmente espacio en blanco.

- Discuta cómo se puede usar esta información para reducir la tasa de datos requerida para una copiadora, como se derivó en el ejemplo 21.1.
- ¿Se pueden lograr ahorros similares en las tasas de datos o espacio de almacenamiento cuando una página (por ejemplo, la página de una revista con fondo negro y fuente blanca) no incluye espacio blanco?

21.10 Tecnologías de impresoras

Con base en la investigación que realice acerca de impresoras de inyección de tinta y láser, compare y contraste las dos tecnologías con respecto a los siguientes atributos:

- Calidad de salida en términos de resolución y contraste.
- Calidad de salida en términos de durabilidad (falta de borrado con el tiempo).
- Latencia y rendimiento total de la impresión.
- Costo de la tinta o toner por página impresa.
- Costo total de propiedad por página impresa.
- Facilidad de uso, que incluye dimensiones físicas, ruido y generación de calor.

21.11 Impresoras de tambor

Un tipo particular de impresora de línea usada hace muchos años era la impresora de tambor. Un gran tambor metálico rotaba a alta rapidez junto a una ruta de papel. El tambor tenía tantas tiras como caracteres en una línea; por decir, 132. En cada tira, las letras y símbolos de interés aparecían en una forma resaltada. También había 132 martillos alineados a lo largo de la longitud del tambor. Cada martillo se controlaba individualmente y golpeaba sobre una cinta justo cuando pasaba por abajo el carácter deseado para dicha posición.

- ¿Por qué tal impresora tendía a embarrar los caracteres?
- ¿Cómo se relaciona la rapidez de impresión de una impresora de tambor, con la latencia de acceso a memoria de disco? En particular, compare con los discos de cabeza por pista y múltiples cabezas.
- Analice las latencias de peor caso y caso promedio para imprimir una línea.

21.12 Detección y medición vía fotoceldas

Una aplicación de las fotoceldas en las fábricas está en la clasificación de objetos que se mueven en una banda transportadora. Si supone que los objetos no se apilan y que no se traslapan en longitud, en la banda transportadora:

- Especifique cómo puede usarse una sola fotocelda para clasificar objetos por longitud. Establezca claramente todas sus suposiciones.
- Proponga un esquema para clasificar objetos de altura fija por sus alturas.
- Repita la parte b), pero esta vez suponga que la altura puede variar a lo largo del objeto.
- Proponga un esquema para clasificar objetos en cubos, esferas y pirámides.

21.13 Motores de rapidez gradual

Un motor de rapidez gradual rota 15° por cada pulso de control recibido. Especifique los tipos de mecanismo que necesitaría si tuviese que usar este motor para controlar el movimiento lateral de la cabeza de impresión para una impresora de inyección de tinta de 600 dpi. Observe que los mecanismos requeridos deben convertir el movimiento rotacional del motor a movimiento lineal con granos más finos.

21.14 Dispositivos de entrada primitivos

Los conmutadores basculantes y *jumpers* constituyen primitivos mecanismos de entrada para dispositivos que no tenían un teclado u otros dispositivos de entrada. Un conmutador basculante puede fijar un bit de entrada a 0 o 1. Un *jumper* hace lo mismo al conectar la entrada a un voltaje constante que representa 0 o 1. Por lo general, se usan cuando se necesitan pocos bits de datos de entrada y la entrada cambia de manera irregular.

- Identifique dos aplicaciones en las que se usen tales dispositivos de entrada.
- Compare los conmutadores basculantes y *jumpers* en relación con la facilidad de uso y flexibilidad.
- Describa cómo ocho conmutadores basculantes y un botón pueden ofrecer una forma de ingresar más de ocho bits de datos.

21.15 Dispositivos especiales de entrada

Investigue los siguientes temas en relación con los dispositivos especiales de entrada y escriba un informe de dos páginas acerca de cada uno.

- Por qué los números impresos al fondo de la mayoría de los cheques bancarios (incluidos el número ID del banco y el número de cuenta) tiene formas inusuales.

- Cómo el código de producto universal (UPC), que se encuentra en la mayoría de los productos, codifica datos y cómo los lee mediante un escáner en el contador de salida.
- Cómo las tarjetas perforadas de 80 columnas (conocidas como Hollerith) codificaban datos y cómo se compara su densidad de almacenamiento, en bits por centímetro cúbico, con los discos y otros tipos de memoria actuales.
- Cómo funcionan los lectores de tarjetas de crédito, que se encuentran en muchas tiendas, cajeros automáticos y estaciones de servicio para la venta de gasolina.
- Cómo el texto escrito a mano, que se ingresa mediante un estilete, se percibe y acepta en las PDA o PC tablet.
- Cómo se comunican con la computadora los teclados, ratones y controles remoto inalámbricos (para cambiar diapositivas en una presentación).

21.16 Dispositivos de salida especiales

Investigue los siguientes temas en relación con los dispositivos de salida especiales y escriba un informe de dos páginas acerca de cada uno.

- Cómo se produce la salida Braille para usuarios invidentes.
- Cómo las impresoras pequeñas, unidas a las calculadoras que imprimen, difieren de las impresoras ordinarias de inyección de tinta o láser.
- Cómo funcionan los dispositivos de presentación por proyección trasera.
- Cómo funciona una pantalla de segmento de línea (por ejemplo, la pantalla LED de siete segmentos para números) y por qué ya no son de uso extenso.

REFERENCIAS Y LECTURAS SUGERIDAS

- | | |
|----------|--|
| [BarC01] | Bar-Cohen, Y., <i>Electroactive polymer (EAP) actuators as artificial muscles: reality, potential, and challenges</i> , SPIE Monograph, vol. PM98, 2001. |
| [Clem00] | Clements, A., <i>The Principles of Computer Hardware</i> , Oxford University Press, 2000. |
| [Howa92] | Howard, W. E., "Thin-Film-Transistor/Liquid Crystal Display Technology: An Introduction", <i>IBM J. Research and Development</i> , vol. 36, núm. 1, pp. 3-10, enero de 1992. |
| [Mint98] | Mintchell, G. A., "Networked I/O Strategies Connect", <i>Control Engineering International</i> , noviembre de 1988. www.manufacturing.net/ctl/index.asp |
| [Myer02] | Myers, R. L., <i>Display interfaces: fundamentals and standards</i> , Wiley, 2002. |

■ CAPÍTULO 22

PROGRAMACIÓN DE ENTRADA/SALIDA

“Lo que anticipamos, rara vez ocurre; lo que menos se espera, generalmente ocurre.”

Benjamin Disraeli

“Se me proporcionó una entrada adicional que era radicalmente diferente de la verdad. Yo ayudé a fomentar dicha versión.”

Coronel Oliver North, de su testimonio Irán-Contra

TEMAS DEL CAPÍTULO

- 22.1** Rendimiento I/O y *benchmarks*
- 22.2** Direccionamiento entrada/salida
- 22.3** I/O calendarizado: sondeo
- 22.4** I/O con base en petición: interrupciones
- 22.5** Transferencia de datos I/O y DMA
- 22.6** Mejora del rendimiento I/O

La entrada y salida, al igual que otras actividades dentro de la computadora, están controlados por la ejecución de ciertas instrucciones dentro de un programa. Mientras que las primeras computadoras tenían instrucción I/O especiales, las máquinas modernas tratan a los dispositivos I/O como si ocuparan parte del espacio de dirección de memoria (I/O mapeado por memoria). Por tanto, la entrada de datos se realiza mediante la lectura de localidades de memoria hasta su acuerdo que se asignan a dispositivos de entrada específicos. En este capítulo, después de mostrar detalles del direccionamiento I/O mediante ejemplos, se revisan los esquemas complementarios de sondeo e interrupciones para sincronizar I/O con el resto de las acciones de un programa. También se discute el rendimiento I/O y cómo se puede mejorar.

■ 22.1 Rendimiento I/O y *benchmarks*

Además de las razones obvias de cargar texto de programa y datos en memoria y grabar los resultados de cálculos en copia impresa y otros formatos, la entrada/salida se puede realizar por una diversidad de razones. Los ejemplos incluyen los siguientes:

Recopilación de datos a partir de redes de sensores.

Actuación de brazos robóticos u otros ensambles en plantas.

Consulta (*querying*) o actualización de bases de datos.

Respaldo de conjuntos de datos o documentos.

Creación de puntos de verificación para evitar reinicio en el evento de una caída.

Con la rápida mejora del rendimiento de CPU durante las dos décadas pasadas, que se proyecta continúe al mismo paso en lo futuro, el rendimiento I/O tomó el escenario central. Como con la “pared de memoria”, discutida en la sección 17.3, las computadoras modernas experimentan una “pared de I/O” que reduce y a la vez nulifica por completo los beneficios del rendimiento del CPU mejorado.

Ejemplo 22.1: La pared de entrada/salida Una aplicación de control industrial pasaba 90% de su tiempo en operaciones de CPU y 10% en I/O cuando se desarrolló originalmente a principios de la década de 1980. Desde entonces, el componente de CPU del sistema se sustituyó con un modelo más nuevo cada cinco años, pero los componentes I/O permanecieron iguales. Si supone que el rendimiento de CPU aumentó por un factor de 10 con cada actualización, derive la fracción de tiempo empleada en I/O durante la vida del sistema.

Solución: Esto requiere la aplicación de la ley de Amdahl, con 90% de la tarea acelerada por 10, 100, 1 000 y 10 000 durante 20 años. Las actualizaciones de CPU redujeron sucesivamente el tiempo de ejecución original de 1 a $0.1 + 0.9/10 = 0.19$, 0.109 , 0.1009 y 0.10009 , lo que hace la fracción de tiempo empleado en operaciones entrada/salida $100 \times (0.1/0.19) = 52.6$, 91.7 , 99.1 y 99.9% , respectivamente. Observe que el último par de actualizaciones CPU no logra mucho respecto de términos de rendimiento mejorado.

A diferencia del rendimiento del CPU, que se puede modelar o estimar con base en características de aplicación y sistema (por ejemplo, mezcla de instrucciones, tasa de fallos de caché, etc.), el rendimiento I/O es bastante difícil de predecir. Esto se debe, en gran medida, a los muchos elementos diferentes involucrados en operaciones I/O: desde el sistema operativo y las unidades de dispositivo a través de los diversos buses y los controladores I/O, hasta los dispositivos I/O en sí. Las interacciones de estos elementos y su contención en usar recursos compartidos como la memoria, se agregan a la dificultad en el modelado, incluso cuando cada elemento individual (bus, memoria, disco) se pueda modelar con precisión.

El rendimiento de entrada/salida se mide de diversas formas, dependiendo de los requisitos de aplicaciones. La *latencia de acceso a I/O* constituye la cabecera de tiempo para una sola operación I/O, que es importante para pequeñas transacciones I/O del tipo que se encuentra en la banca o comercio electrónicos. La *tasa de datos I/O*, con frecuencia expresada en megabytes por segundo, es relevante a las grandes transferencias de datos usualmente de interés en aplicaciones de supercomputadoras. El *tiempo de transferencia de datos I/O* se relaciona con el tamaño del bloque que se debe transferir y la tasa de datos. El *tiempo de respuesta* representa la suma de la latencia de acceso y el tiempo de transferencia de datos; su inverso produce el número de operaciones I/O por unidad de tiempo (*rendimiento total I/O*). El tiempo de respuesta y el rendimiento total I/O con frecuencia se negocian uno contra otro. Por ejemplo, el procesamiento fuera de orden de los accesos a disco hace que la latencia sea muy variable, y lo incrementa en muchos casos, pero puede mejorar el rendimiento total significativamente.

En proporción con las diversas características de I/O en diferentes áreas de aplicación, existen tres tipos de *benchmark I/O*:

1. Los *benchmarks I/O* de supercomputadoras se enfocan en leer grandes volúmenes de datos de entrada, escribir muchas instantáneas para crear puntos de verificación de cálculos muy largos

y salvar un conjunto relativamente pequeño de resultados de salida al final. Aquí el parámetro clave es el rendimiento total de datos I/O, expresado en megabytes por segundo. La latencia I/O es menos importante, en tanto se pueda mantener un rendimiento total elevado.

2. Los *benchmarks* I/O de procesamiento de transacción usualmente tratan con una base de datos enorme, pero cada transacción es bastante pequeña, ello involucra algunos accesos a disco (por decir, 2-10), con unas cuantas miles de instrucciones ejecutadas por acceso a disco. En concordancia, el parámetro importante en tales *benchmarks* es la tasa I/O, expresada en número de accesos a disco por segundo.
3. Los *benchmarks* I/O de sistema de archivo se enfocan en accesos a archivo (operaciones de lectura y escritura), creación de archivos, gestión de directorio, indexación y reestructuración. Dadas las variadas características de los archivos y accesos a archivo en diferentes aplicaciones, con frecuencia tales *benchmarks* están especializados a un dominio de aplicación particular (por ejemplo, cálculos científicos).

Cada una de estas categorías se puede refinar aún más. Por ejemplo, el procesamiento de transacción abarca un espectro de dominios de aplicación, desde las simples interacciones web hasta complejas solicitudes que involucran gran cantidad de procesamiento. Como con los *benchmarks* de CPU, discutidos en la sección 4.4, se puede incorporar una mezcla de características en un *benchmark* I/O sintético o real para valorar un sistema para multidominio o uso de propósito general.

22.2 Direccionamiento entrada/salida

En la I/O mapeada por memoria, cada dispositivo de entrada o salida tiene uno o más registros de hardware desde/hacia los que se puede leer/escribir, como si fuesen ubicaciones de memoria. Por tanto, no se necesitan instrucciones I/O especiales y se puede cargar y almacenar instrucciones para efectuar I/O. Como ejemplos en MiniMIPS se consideran la entrada por teclado y la salida en pantalla.

Como se muestra en la figura 22.1, la unidad de teclado en MiniMIPS tiene un par de localidades de memoria asociadas con un registro de control de 32 bits (dirección `0xffff0000`) y un registro de datos (`0xffff0004`). El registro de control retiene varios bits de estatus que portan información acerca del estatus del dispositivo y modos de transmisión de datos. Dos bits en el registro de control que son relevantes para la discusión son el indicador R de “dispositivo listo” y la bandera (*flag*) IE de “habilitar

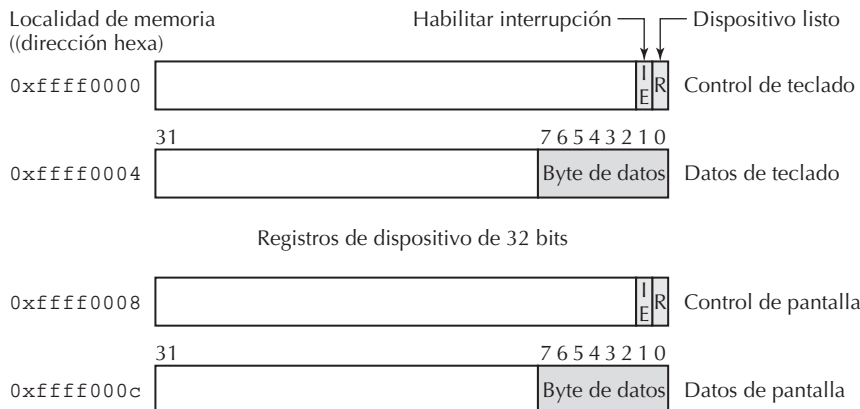


Figura 22.1 Registros de control y datos para teclado y unidad de pantalla en MiniMIPS.

interrupción” en las posiciones de bit 0 y 1, respectivamente (el uso de R se discutirá dentro de poco; IE se necesitará en el capítulo 24). El registro de datos puede contener un símbolo ASCII en su byte de extrema derecha. Cuando una tecla se presiona en el teclado, la lógica de éste determina cuál símbolo se enviará a la computadora y lo coloca en el byte más bajo del registro de datos del teclado, además postula el bit R para indicar que un símbolo está listo para transmisión. Si el programa carga la palabra de control de teclado desde la localidad de memoria 0xffff0000 en algún registro, aprenderá, al examinar el bit 0, que el teclado tiene un símbolo para transmitir. En este sentido, el programa puede cargar el contenido de la localidad 0xffff0004 en un registro para determinar cuál símbolo se contiene en el registro de datos del teclado. La lectura del registro de datos del teclado hace que R se despostule automáticamente. Observe que R es un bit de sólo lectura en el sentido de que si el procesador escribe algo en el registro de control de teclado el estado de R no cambiará.

Ejemplo 22.2: Entrada de datos desde teclado Escriba una secuencia de instrucciones en lenguaje ensamblador MiniMIPS para hacer que el programa espere hasta que el teclado tenga un símbolo para transmitir y luego lea el símbolo en el registro \$v0.

Solución: El programa debe examinar continuamente el registro de control de teclado para determinar si R se postuló. Finalmente, cuando el bit R se encuentre postulado, termina la inactividad del programa (también conocida como *busy wait*, ciclo de espera) y el símbolo en el registro de datos de teclado se carga en \$v0.

```

        lui    $t0, 0xffff      # poner 0xffff0000 en $t0
idle:   lw     $t1, 0($t0)       # obtiene palabra control del teclado
        andi   $t1, $t1, 0x0001 # aísla el LSB (bit R)
        beq    $t1, $zero, idle # si no listo (R = 0), esperar
        lw     $v0, 4($t0)      # recuperar palabra datos desde
                                # teclado

```

Este tipo de entrada es adecuada sólo si la computadora espera alguna entrada crítica y no puede hacer nada útil en ausencia de esta entrada.

De igual modo, la unidad de representación visual (pantalla) en MiniMIPS tiene un par de localidades de memoria asociadas con su registro de control de 32 bits (localidad 0xffff0008) y registro de datos (0xffff000c), como se muestra en la figura 22.1. El registro de control de pantalla es similar al del teclado, excepto que el bit R indica que la unidad de pantalla está lista para aceptar un nuevo símbolo en su registro de datos. Cuando un símbolo se copia en este registro de datos, R se despostula automáticamente. Si el programa carga la palabra control de pantalla desde la localidad de memoria 0xffff0008 en algún registro, aprenderá, al examinar el bit 0, que la unidad de pantalla está lista para aceptar un símbolo. En consecuencia, el programa puede almacenar el contenido de un registro en la localidad 0xffff000c, y envía un símbolo a la unidad de pantalla.

Ejemplo 22.3: Salida de datos a unidad de pantalla Escriba una secuencia de instrucciones en lenguaje ensamblador MiniMIPS para hacer que el programa espere hasta que la unidad de pantalla esté lista de modo de que acepte un nuevo símbolo, luego escriba el símbolo del registro \$a0 al registro de datos de la unidad de pantalla.

Solución: El programa debe examinar continuamente el registro de control de la unidad de pantalla para determinar si el bit R se postuló. Eventualmente, cuando el bit R se encuentra postulado, el símbolo en el registro \$a0 se copia en el registro de datos de la unidad de pantalla.

```

        lui    $t0, 0xffff      # poner 0xffff0000 en $t0
idle:   lw     $t1, 8($t0)       # obtener palabra control de pantalla
        andi   $t1, $t1, 0x0001 # aísla el LSB (bit R)
        beq    $t1, $zero, idle # si no está listo (R = 0), esperar
        sw     $a0, 12($t0)     # suministrar datos de palabra a
                                # unidad de pantalla

```

Este tipo de salida es adecuado sólo si se es costeable tener el CPU dedicado a transmisión de datos a la unidad de pantalla.

El hardware proporcionado en cada controlador de dispositivo I/O para permitir este tipo de direccionamiento es muy simple. Como se muestra en la figura 22.2, el controlador de dispositivo se conecta al bus de memoria. La dirección de memoria que se observa en el bus se compara continuamente contra la propia dirección del dispositivo almacenada en un registro interno. Si esta dirección de dispositivo está en un registro de sólo lectura, el dispositivo tiene una *dirección dura* (no modificable); de otro modo, el dispositivo tiene una *dirección blanda* (modificable). Cuando se detecta una equiparación de dirección, el contenido del registro de estatus o datos se coloca en (lee), o carga desde (escribe), las líneas de datos del bus.

Observe que las nociones discutidas en esta sección se requieren sólo si resulta necesario escribir controladores de dispositivo para computadoras de propósito general o para realizar operaciones I/O en un ambiente de sistema incrustado específico de aplicación. La mayoría de los usuarios están protegidos contra tales detalles porque sus operaciones I/O se realizan a un nivel bastante elevado a través de peticiones al sistema operativo. Por ejemplo, en el simulador SPIM para MiniMIPS, el usuario o compilador coloca un código apropiado en el registro \$v0 (y uno o más de otros parámetros en ciertos registros con los que se concuerda) y luego ejecuta una instrucción de petición de sistema (`syscall`). Para detalles, vea la sección 7.6, en particular la tabla 7.2.

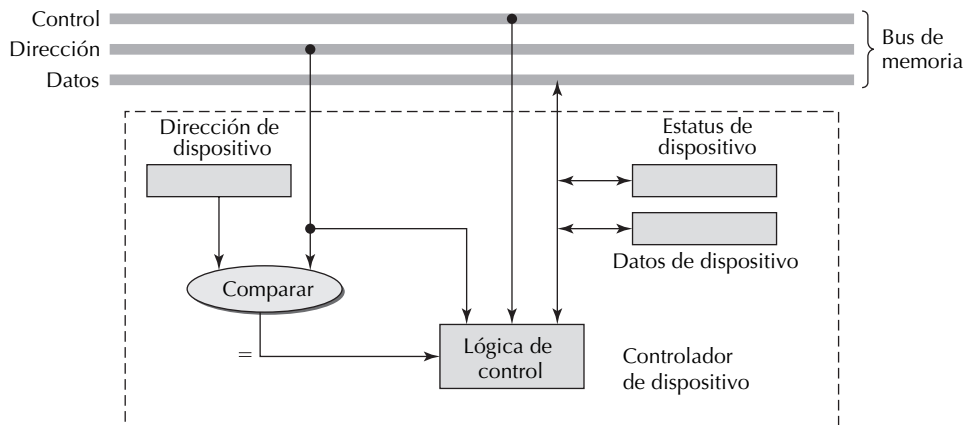


Figura 22.2 Lógica de direccionamiento para un controlador de dispositivo I/O.

■ 22.3 I/O calendarizado: sondeo

Los ejemplos 22.2 y 22.3 representan instancias de I/O sondeadas: el procesador inicia la I/O al preguntar al dispositivo si está listo para enviar/recibir datos. Con cada interacción, se transfiere una sola unidad de datos. Si el dispositivo no está listo, el procesador no necesita esperar en un ciclo ocupado; más bien, puede realizar otras tareas y verificar más tarde con el dispositivo. Ningún problema surge en tanto el procesador verifica el dispositivo I/O con suficiente frecuencia para garantizar que no hay pérdida de datos.

Ejemplo 22.4: Entrada a través de sondear un teclado Un teclado se debe interrogar al menos diez veces por segundo para garantizar que ninguna tecla fue presionada por el usuario y se haya perdido. Suponga que cada una de tales interrogaciones y transferencia de datos asociados tarda 800 ciclos de reloj en un procesador con un reloj de 1 GHz. ¿Qué fracción del tiempo CPU se gasta en sondear el teclado?

Solución: La fracción tiempo CPU empleado para sondear el teclado se obtiene al dividir el número de ciclos necesarios para diez interrogaciones por el número total de ciclos disponibles en un segundo:

$$(10 \times 800)/10^9 \cong 0.001\%$$

Observe que la tasa de sondeo de diez veces por segundo, o 600 veces por minuto, se eligió para ser mayor que incluso la velocidad del mecanógrafo más rápido, de modo que no hay oportunidad de perder un carácter que se ingrese, y luego se sobrescriba, en el *buffer* de datos del teclado.

A partir del ejemplo 22.4 se ve que el teclado es un dispositivo de entrada lento que el procesador puede conservar con él fácilmente. Después de cada sondeo del teclado, el procesador tiene unos $10^9/10 - 800 \cong 10^8$ ciclos de reloj (0.1 s) para atender a otras tareas antes de que tenga que sondear de nuevo el teclado. Los siguientes dos ejemplos muestra que otros dispositivos I/O son más demandantes a este respecto.

Ejemplo 22.5: Entrada a través de sondear una unidad de disco flexible Una unidad de disco flexible envía o recibe datos cuatro bytes a la vez y tiene una tasa de datos de 50 KB/s. Para garantizar que los datos antiguos no se sobrescriben con los nuevos en el *buffer* de dispositivo durante una operación de entrada, el procesador debe muestrear el *buffer* a la tasa de $50K/4 = 12.5K$ veces por segundo. Suponga que cada interrogación y transferencia de datos asociados tarda 800 ciclos de reloj en un procesador con un reloj de 1 GHz. ¿Qué fracción del tiempo de CPU se emplea en sondear la unidad de disco flexible?

Solución: La fracción de tiempo CPU empleado en sondear la unidad de disco flexible se obtiene al dividir el número de ciclos necesarios para 12.5K interrogaciones por el número total de ciclos disponibles en un segundo:

$$(12.5K \times 800)/10^9 = 1\%$$

Observe que la cabecera en tiempo CPU para sondear la unidad de disco flexible es mil veces el del sondeo del teclado en el ejemplo 22.4.

Ejemplo 22.6: Entrada a través de sondear una unidad de disco duro Una unidad de disco duro transfiere datos cuatro bytes a la vez y tiene una tasa de datos pico de 3 MB/s. Para garantizar que datos antiguos no se sobrescriben con nuevos datos en el *buffer* de dispositivo durante una operación de entrada, el procesador debe muestrear el *buffer* a la tasa de $3\text{M}/4 = 750\text{K}$ veces por segundo. Suponga que cada interrogación y transferencia de datos asociados tarda 800 ciclos de reloj en un procesador con un reloj de 1 GHz. ¿Qué fracción del tiempo CPU se emplea en sondear la unidad de disco duro?

Solución: La fracción del tiempo CPU que se emplea en sondear la unidad de disco duro se obtiene al dividir el número de ciclos necesarios para 750K interrogaciones por el número total de ciclos disponible en un segundo:

$$(750\text{K} \times 800)/10^9 = 60\%$$

Un solo dispositivo I/O que ocupa 60% de tiempo CPU es inaceptable.

Observe que el disco duro del ejemplo 22.6 mantiene el CPU casi completamente ocupado: entre dos interrogaciones consecutivas, el CPU sólo tiene $10^9/750\,000 - 800 \cong 533$ ciclos de reloj ($0.5\ \mu\text{s}$) para atender otras tareas. Esto último puede no ser suficiente para realizar mucho trabajo útil. La unidad de disco flexible del ejemplo 22.4 es intermedia entre el teclado muy lento y la unidad de disco duro muy rápida. Deja al CPU con $10^9/12\,500 - 800 = 79\,200$ ciclos de reloj ($79\ \mu\text{s}$) de tiempo disponible entre interrogaciones consecutivas.

22.4 I/O con base en petición: interrupciones

Gran cantidad de tiempo CPU se desperdicia en sondear cuando el dispositivo está completamente inactivo o todavía no está listo para enviar o recibir datos. Incluso los dispositivos lentos pueden provocar cabecera inaceptable cuando hay muchos de estos dispositivos que deben sondearse de manera continua. Sondear cientos o miles de sensores y actuadores en un escenario de control industrial puede ser muy derrochador. Por ejemplo, un detector de temperatura no necesita enviar datos a la computadora a menos que haya un cambio que supere un umbral predefinido o que se alcance cierta temperatura crítica. En la I/O activada por interrupción, el dispositivo inicia la I/O al enviar una señal de interrupción al CPU. Cuando éste recibe una señal de interrupción, transfiere el control a una rutina de interrupción especial que determina la causa de la interrupción e inicia acción apropiada.

Ejemplo 22.7: Entrada basada en interrupción desde una unidad de disco duro Considere el mismo disco que en el ejemplo 22.6 (transferencia de “trozos” de 4 B de datos a 3 MB/s cuando está activo) y suponga que el disco está activo más o menos 5% del tiempo. La cabecera de interrupción al CPU y la realización de la transferencia es de casi 1 200 ciclos de reloj de un procesador de 1 GHz. ¿Qué fracción del tiempo de CPU se emplea en atender la unidad de disco duro?

Solución: A partir del ejemplo 22.6 se sabe que 750K interrupciones ocurrirán en cada segundo cuando la unidad de disco esté activa, dados la tasa de datos del disco y la transmisión de datos en “trozos” de 4 B. La fracción de tiempo de CPU empleada en interrupciones desde el disco duro se obtiene al dividir

el número de ciclos necesarios para atender el número esperado de interrupciones por segundo por el número total de ciclos disponibles en un segundo:

$$0.05 \times (750K \times 1\,200)/10^9 = 4.5\%$$

En este contexto, aun cuando la cabecera del I/O activado por interrupción sea mayor cuando el disco esté activo, pues el disco usualmente está inactivo, el tiempo CPU global empleado en I/O es mucho menor.

Las interrupciones se controlan mediante bits de estatus en el lado del dispositivo (solicitante) y en el lado del CPU (proveedor de servicio). El bit de “habilitar interrupción” (IE) en los registros de control de la figura 22.1 dice al dispositivo que debe enviar una señal de interrupción al CPU cuando se postule el bit R de “dispositivo listo”. El lado CPU tiene una bandera correspondiente que indica si las interrupciones del teclado, unidad de pantalla u otros dispositivos serán aceptados. Si las interrupciones se habilitan, la señal de interrupción se reconoce y se ingresa un protocolo que conduce a que los datos se acepten desde, o se envíen a, el dispositivo solicitante. Las interrupciones que no se habilitan se dice que están *enmascaradas*. Usualmente, las interrupciones se enmascaran o deshabilitan sólo durante cortos periodos cuando el CPU debe atender funciones críticas sin interrupción.

Hasta la detección de una señal de interrupción, siempre que la interrupción particular o clase de interrupción no se enmascaren, el CPU reconoce la interrupción (de modo que el dispositivo puede despostular su señal de solicitud) y comienza a ejecutar una rutina de servicio de interrupción. Para servir una solicitud de interrupción se realiza el siguiente procedimiento:

1. Salvar el estado de CPU y pedir la rutina de servicio de interrupción.
2. Deshabilitar todas las interrupciones.
3. Salvar información mínima acerca de la interrupción en la pila.
4. Habilitar interrupciones (o al menos las de mayor prioridad).
5. Identificar la causa de la interrupción y atender la solicitud subyacente.
6. Restaurar el estado del CPU a lo que existía antes de la última interrupción.
7. Regresar de la rutina de servicio de interrupción.

La cabecera de cada solicitud de interrupción es mayor que la del sondeo porque los pasos requeridos para la última corresponden a una parte del paso 5; los otros pasos, o la identificación de la causa de interrupción en el paso 5, no se necesitan con sondeo. Observe que las interrupciones se habilitan en el paso 5 antes de que la manipulación de la interrupción actual esté completa. Esto es así porque deshabilitar las interrupciones durante periodos prolongados puede conducir a pérdida de datos o ineficiencias en entrada/salida. Por ejemplo, una cabeza de lectura/escritura puede pasar el sector deseado mientras las interrupciones están deshabilitadas, lo que fuerza al menos una revolución adicional, con su latencia acompañante. La capacidad de manipular *interrupciones anidadas* es importante al tratar con múltiples dispositivos I/O de alta rapidez o aplicaciones de control sensibles al tiempo. Las interrupciones se discutirán con mayor detalle en el capítulo 24.

■ 22.5 Transferencia de datos I/O y DMA

Observe que, si los datos se transfiriesen entre la memoria principal y el dispositivo I/O en “trozos” más grandes, en lugar de unidades de 1-4 bytes supuestas en los ejemplos del 22.4 al 22.7, la cabecera de interrupciones se reduciría y se despreciaría menos tiempo de CPU. Lo último en esta dirección es dejar que el CPU inicie una operación I/O (por su propia iniciativa o al recibir una interrupción) y dejar que un controlador I/O inteligente copie los datos desde un dispositivo de entrada hacia memoria o

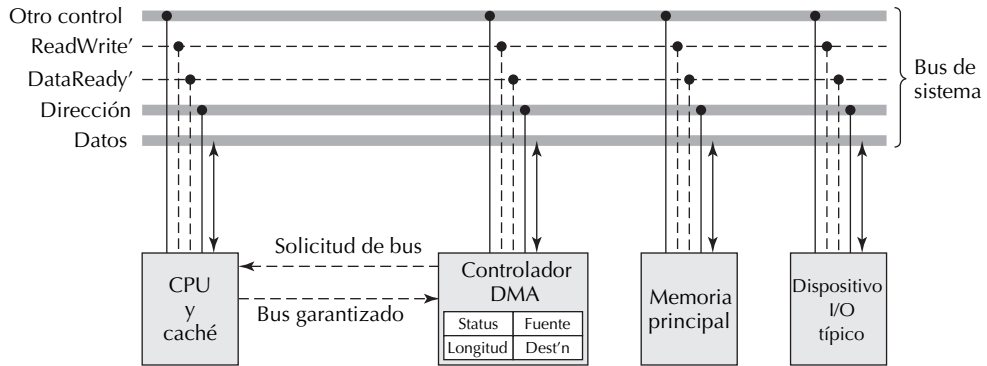


Figura 22.3 El controlador DMA comparte el bus de sistema o memoria con el CPU.

desde memoria hacia un dispositivo de salida. Este enfoque se denomina entrada/salida DMA (por sus siglas en inglés, *direct memory access*, acceso directo a memoria).

Un controlador DMA representa un procesador simple que puede adquirir el control del bus de memoria del CPU y luego actuar como el CPU lo haría para controlar transferencias de datos entre dispositivos I/O y memoria. El CPU interactúa con el controlador DMA en gran parte como lo hace con un dispositivo I/O, excepto que sólo se intercambia información de control. Al igual que el controlador de dispositivo de la figura 22.2, el controlador DMA tiene un registro de status que se usa para comunicar información de control con el CPU. Otros tres registros sustituyen el registro de datos de dispositivo de la figura 22.2:

1. Registro dirección-fuente, donde el CPU coloca una dirección para la fuente de datos.
2. Registro dirección-destino, que retiene el destino para la transferencia de datos.
3. Registro de longitud, donde el CPU coloca el número de palabras de datos a transferir.

La figura 22.3 muestra la relación del controlador DMA con el CPU, la memoria y los controladores de dispositivo, si se supone el uso de un solo bus de sistema para accesos a memoria e I/O. En la práctica, la mayoría de los dispositivos I/O se conectan a buses separados que se vinculan al bus de memoria a través de adaptadores o puentes (figura 21.2). Sin embargo, en la discusión del DMA, se procede con la suposición de un solo bus de sistema, como en la figura 21.1.

Con base en la información colocada por el CPU en varios registros de un controlador DMA, el último puede tomar el control y realizar los pasos de una transferencia de datos preescrita desde un dispositivo de entrada hacia memoria o desde memoria hacia un dispositivo de salida en forma muy parecida a como lo haría el CPU. Observe que el uso de DMA es compatible con instrucciones I/O tanto mapeadas por memoria como para especiales. De hecho, la memoria y los dispositivos I/O no pueden decir si el CPU o el controlador DMA controlan las transacciones de bus, pues, en cualquier caso, se sigue el mismo protocolo de bus.

Normalmente el CPU está a cargo del bus de memoria y supone que éste se encuentra disponible para su uso en cualquier momento. Por tanto, antes de que el controlador DMA use el bus, postula una señal de *solicitud de bus* que informa al CPU de su intención. Los circuitos de control del bus en el CPU observan esta solicitud y, cuando es conveniente para el CPU, postulan una señal de *bus garantizado*. El controlador DMA ahora puede hacerse cargo del bus y realizar la transferencia de datos solicitada. Cuando la transferencia de datos se completa, el controlador DMA despostula la señal de solicitud de bus y el CPU, a su vez, despostula la señal de bus garantizado, ello pone al bus de vuelta bajo su

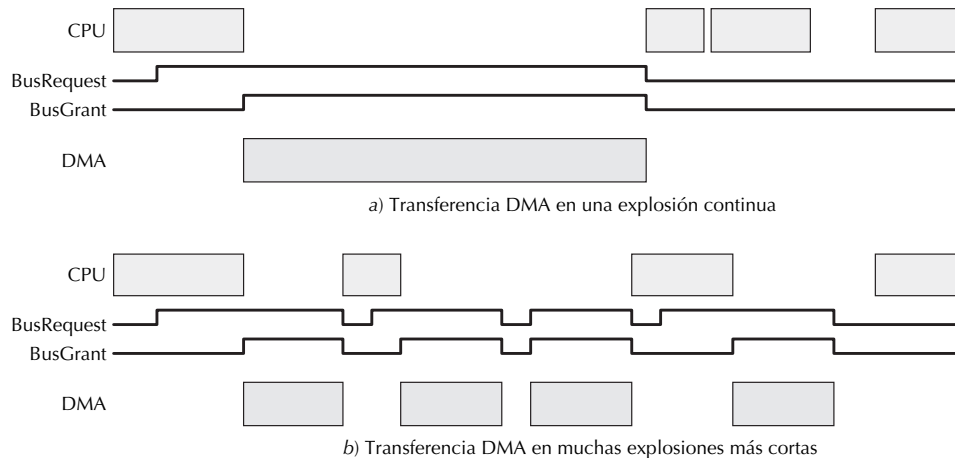


Figura 22.4 Operación DMA y las transferencias asociadas de control de bus.

control (figura 22.4a). En consecuencia, las señales tanto de solicitud de bus como de bus garantizado permanecen postuladas durante el uso del bus por parte del controlador DMA.

Durante una transferencia DMA, el CPU puede continuar ejecutando instrucciones, en tanto que hacerlo no involucra transferencia de datos sobre el bus de memoria. Esto es muy factible con los CPU modernos, dada la alta tasa de impacto en la caché (o cachés combinadas L1 y L2) usualmente integradas en el mismo chip que el CPU. Si, debido a fallos de caché o algún otro evento, el CPU necesita usar el bus de memoria antes de que la transferencia DMA se haya completado, debe esperar hasta que el controlador DMA libere el bus (figura 22.4a).

Para evitar que el CPU este inactivo durante mucho tiempo, lo que conduce a una degradación de rendimiento significativa, la mayoría de los controladores DMA se diseñan para ser capaces de romper una transferencia de datos I/O larga en muchas transferencias más cortas que involucran pocas palabras. Cuando el controlador DMA adquiere el bus, transfiere un número prefijado de palabras antes de renunciar al bus. Después de un breve retardo, el controlador DMA solicita el bus nuevamente para la siguiente etapa de su transferencia de datos. Mientras tanto, el CPU vuelve a ganar el control del bus y puede usarlo para leer (*fetch*) palabras de memoria que permitirían continuar su operación. Este tipo de bloque o ciclo de transferencia de datos DMA se muestra en la figura 22.4b.

Ejemplo 22.8: Entrada basada en DMA desde una unidad de disco duro Considere la unidad de disco duro de los ejemplos 22.6 y 22.7 (con tasa de datos pico de 3 MB/s durante 5% del tiempo cuando está activo). El disco tiene 512 B sectores y rota a 3 600 rpm. Ignore brechas y otras cabeceras en el almacenamiento de datos en las pistas del disco. ¿Cómo se compara la transferencia DMA, en términos de cabecera de tiempo para el CPU de 1 GHz, con el sondeo y la I/O basada en interrupciones de los ejemplos 22.6 y 22.7? Suponga que se necesitan 800 ciclos de reloj para establecer una transferencia de datos DMA y 1 200 ciclos de reloj para manipular la interrupción generada en la conclusión y que cada transferencia de datos involucra a) un sector, o b) toda una pista.

Solución: La capacidad de pista de disco se puede obtener a partir de la tasa de datos de disco y su rapidez de rotación: $(3 \text{ MB/s}) / (60 \text{ revoluciones/s}) = 0.05 \text{ MB/revolución}$, o una capacidad de pista de 50 KB = 100 sectores. El CPU pasa $800 + 1\,200 = 2\,000$ ciclos de reloj, o 2 μs , para establecer y procesar

la terminación de cada transferencia de datos. Considere primero las transferencias de datos de sector. Si el disco transfiere activamente datos 5% del tiempo, debe leer $0.05 \times 3 \text{ MB/s} = 150 \text{ KB/s} = 300$ sectores/s. La cabecera de tiempo para el CPU es, por tanto, $2 \mu\text{s} \times 300 = 0.6 \text{ ms}$ (0.06%). Esto último se compara muy favorablemente con las cabeceras de sondeo (60%) y transferencias palabra por palabra controladas por CPU (4.5%) derivadas en los ejemplos 22.6 y 22.7. Si todas las pistas se transfieren, y la actividad del disco permanece a 5%, debe haber un factor de 100 transferencias de datos menos, ello conduce a una reducción por el mismo factor en cabecera de CPU. Observe que el paso repetido de control del bus entre CPU y controlador DMA involucra cierta cabecera que se ignoró en este ejemplo. Esta última cabecera es más significativa cuando cada transferencia de datos se realiza en la forma cíclica que se muestra en la figura 22.4b. Sin embargo, esta cabecera no es suficiente para minimizar seriamente los beneficios de I/O basada en DMA.

Observe que es posible tener más de un controlador DMA compartiendo el bus de memoria con el CPU. En tal caso, así como cuando hay múltiples CPU, un mecanismo de arbitraje de bus toma el lugar de la señalización de garantía de petición entre un controlador DMA y un solo CPU. El arbitraje de bus se discutirá en la sección 23.4.

A pesar de la clara ventaja en rendimiento de I/O basado en DMA, este método no está libre de problemas y engaños. Por ejemplo, ciertos problemas de implementación surgen cuando las interacciones de DMA involucran ubicaciones en el espacio de dirección de memoria, los problemas relacionados con el uso de direcciones físicas o virtuales para especificar la fuente o destino de las transferencias y la parte de la jerarquía de memoria desde (hacia) la que se toman (copian) los datos. He aquí una lista de problemas que necesitan considerarse.

1. *DMA que usan direcciones físicas:* Puesto que páginas virtuales consecutivas no necesariamente son contiguas en memoria principal, una transferencia multipágina de este tipo no se puede especificar con una dirección y longitud bases; por tanto, las transferencias de datos más largas se deben romper en algunas transferencias de la misma página.
2. *DMA que usan direcciones virtuales:* el controlador DMA necesita traducción de dirección.
3. *DMA que escribe en, o lee desde, memoria principal:* el CPU puede acceder a datos añejos en caché, o el controlador DMA puede obtener datos que no estén actualizados.
4. *DMA que opera vía caché:* los datos I/O pueden desplazar datos de CPU activos, esto último propicia degradación del rendimiento.

En cualquier caso, se necesita la cooperación del sistema operativo para asegurar que, una vez proporcionada una dirección al controlador DMA, permanece válida a través de la transferencia de datos (es decir, las páginas no se desplazan en memoria principal).

Es prudente reiterar que todos estos detalles de I/O usualmente no son visibles a los programas de usuario porque inician I/O a través de peticiones de sistema.

■ 22.6 Mejora del rendimiento I/O

Como con la mayoría de los libros acerca de arquitectura de computadoras, la mayor parte de este libro se dedica a métodos para diseñar CPU más rápidos y superar la brecha de rapidez CPU-memoria. Sin embargo, mejorar la rapidez de CPU y memoria sin batallar con problemas en el procesamiento I/O sería de valor limitado o fútil. No es raro que una aplicación pase más tiempo en I/O que en cálculo. Incluso cuando I/O no domine el tiempo de ejecución de una aplicación particular, la ley de Amdahl recuerda que la mejora en el rendimiento esperado, mientras se ignora I/O, es más bien limitada. Por tanto, es impor-

tante reconocer los problemas en entrada y salida y estar atento a los métodos para tratar con ellos. Como se discutió en la sección 22.1, el rendimiento de entrada/salida se mide por varios parámetros que no son independientes uno del otro y que con frecuencia se pueden negociar uno contra otro. Son éstos:

- Latencia de acceso (cabecera de tiempo de una operación I/O).
- Tasa de transferencia de datos (medida en megabytes por segundo).
- Tiempo de respuesta (tiempo consumido desde la solicitud hasta la conclusión).
- Rendimiento total (número de operaciones I/O por segundo).

En consecuencia, en esta sección se discutirán algunos métodos para mejorar cada una de estas facetas de rendimiento I/O. Los métodos abarcan un amplio rango de opciones, desde programas de afinación de aplicación para actividad I/O reducida, hasta proporcionar mejoras arquitectónicas o auxiliares de hardware para manipulación de I/O de cabecera baja.

La latencia de acceso representa la cabecera de tiempo total para una operación I/O. Incluye retardos obvios y bien comprendidos, como el tiempo de búsqueda de disco o la latencia rotacional, así como otras cabeceras que son menos obvias pero pueden ser muy significativas. Tales cabeceras se acumulan. Un ejemplo de cómo toda la cabecera proveniente de muchas capas de hardware y software podrían llevar I/O al frente, se encuentra en una sesión de conexión lógica (*login*) remota en la que los caracteres mecanografiados en un sitio local se deben transmitir a un sistema remoto. El número de eventos y capas involucrados en la transmisión de un carácter o comando desde el sitio local hasta la computadora remota es aturdidor [Silb02]:

<u>Lado que envía (local)</u>	<u>Lado que recibe (remoto)</u>
Escritura de carácter	Recepción de paquete
Generación de interrupción	Adaptador de red
Conmutador de contexto	Generación de interrupción
Manipulador de interrupción	Conmutador de contexto
Controlador dispositivo teclado	Controlador de dispositivo
Kernel de sistema operativo	Extracción de carácter
Conmutación de contexto	Kernel de sistema operativo
Proceso de usuario	Conmutación de contexto
Petición de sistema para salida	Daemon de red
Conmutación de contexto	Conmutación de contexto
Kernel de sistema operativo	Kernel de sistema operativo
Controlador dispositivo de red	Conmutación de contexto
Adaptador de red	Subdaemon de red
Generación de interrupción	
Conmutación de contexto	
Manipulador de interrupción	
Conmutación de contexto	
Conclusión de petición de sistema	

En el lado receptor hay un programa de sistema llamado *daemon* (demonio) de red, que manipula la identificación y asignación de datos de red entrantes y otro programa, el subdaemon, dedicado a manipulación de I/O de red para una sesión de *login* específica. Ahora, si el receptor tiene que repetir (eco) el carácter al que envía, ¡todo el proceso se repite en la dirección inversa! La mejora en cada uno de los pasos o agentes apenas mencionados puede conducir a un mejor rendimiento I/O. Por ejemplo, los

auxiliares de hardware para conmutación de contexto de baja cabecera (a discutir en la sección 24.5) pueden tener un efecto significativo, pues “conmutación de contexto” aparece ocho veces en la lista.

Aun cuando la latencia de acceso en sí no se pueda disminuir, se puede lograr un efecto equivalente al reducir el número de operaciones I/O al aumentar el tamaño de cada operación. Leer la misma cantidad de datos mediante un número pequeño de solicitudes I/O tiene el efecto de reducir la latencia de acceso efectiva por megabyte de lectura de datos.

Ejemplo 22.9: Ancho de banda I/O efectivo de disco Considere una unidad de disco duro con 512 B sectores, una latencia de acceso promedio de 10 ms (incluida cabecera de software) y un rendimiento total pico de 10 MB/s. Grafique la variaciones en el ancho de banda I/O efectivo conforme la unidad de transferencia de datos (bloque) varía en tamaño desde 1 sector (0.5 KB) hasta 1 024 sectores (500 KB). Ignore todas las cabeceras de grabación de datos y brechas intersector.

Solución: Cuando un sector se transfiere por acceso a disco, el tiempo de transferencia de datos de $0.5/10 = 0.05$ (cantidad de transferencia en kilobytes divididos entre tasa de transferencia en kilobytes por milisegundo) sumado a la latencia de acceso de 10 ms produce un tiempo total I/O de 10.05 ms. Lo anterior corresponde a un ancho de banda efectivo de $0.5/10.05 = 0.05$ MB/s. Cálculos similares para tamaños de bloque de 10, 20, 50, 100, 200, 300, 400 y 500 KB producen la tendencia que se muestra en la figura 22.5. Observe que el ancho de banda efectivo mejora de manera más bien rápida conforme el tamaño de bloque aumenta y alcanza la mitad del valor pico teórico en un tamaño de bloque de 100 MB. De ahí en adelante, la mejoría no es tan significativa, ello sugiere que puede que no valga la pena aumentar todavía más el tamaño de bloque, dada la posibilidad de llevar datos menos útiles.

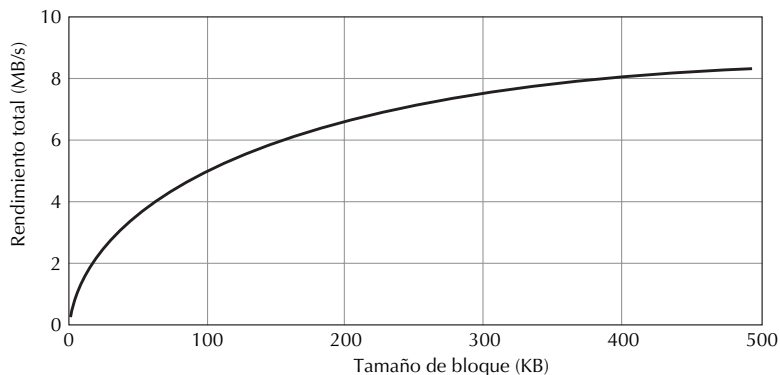


Figura 22.5 Rendimiento total I/O efectivo contra la cantidad de datos transferidos con cada petición I/O.

La tasa de transferencia de datos I/O representa la cantidad de datos que se pueden transferir entre un dispositivo y memoria en unidad de tiempo. Las tasas de transferencia pico citadas para varios dispositivos I/O usualmente no se logran en la práctica. Esto último se debe, en parte, a que los dispositivos pueden tener tasas variables; por ejemplo, pueden diferir las tasas de transferencia desde las pistas más interna y más externa de un disco. Sin embargo, la tasa de transferencia real es una función no sólo de la habilidad del dispositivo para recibir o enviar los datos, sino también de la capacidad de un anfitrión (*host*) de recursos intermedios (controladores, buses, puentes o adaptadores de bus, etc.) para relevar los datos. Observe que, aun cuando un bus pueda ser tan rápido como para manipular la tasa de datos pico desde un disco cuando el tráfico de disco a memoria es el único tipo presente, compartir el bus por otras actividades en marcha puede reducir su habilidad para lidiar con el I/O de disco.

El tiempo de respuesta se define como el tiempo transcurrido desde la emisión de una solicitud I/O hasta su conclusión. En el caso más simple, cuando un dispositivo I/O se dedica a una sola aplicación o usuario, el tiempo de respuesta constituye la suma de latencia de acceso I/O y tiempo de transferencia de datos (cantidad de datos transferidos dividido por tasa de transferencia), en tal caso, el tiempo de respuesta no es una medida independiente sino que es derivable a partir de los dos parámetros previos. Sin embargo, se puede agregar un *retardo por cola* no despreciable cuyo valor depende de la mezcla actual de operaciones I/O pendientes en varios puntos en la ruta de transferencia I/O. Los retardos por cola son difíciles de valorar pero forman el centro de una área de estudio conocida como *teoría de colas*, cuya discusión no compete a este libro. Baste con decir que las colas pueden causar comportamiento I/O inesperado, y a veces contraintuitivo. Una buena analogía la proporciona la cola de los clientes en un banco. Si una cola de diez clientes se acumula durante la hora del almuerzo, incluso si el rendimiento total de servicio del banco es comparable a la tasa de llegada de nuevos clientes, la larga cola puede seguir durante toda la tarde. En otras palabras, aun cuando el rendimiento total no sea un problema en este caso, el tiempo de respuesta es muy malo. Este tipo de efecto de cola se puede experimentar, por ejemplo, en comunicación de red durante periodos de tráfico pico.

Otro factor que tiene un efecto adverso sobre la latencia I/O es la copia repetida de datos I/O de una área de memoria a otra como resultado de las múltiples capas de software que se involucran en la manipulación de una petición I/O. Por ejemplo, los sistemas operativos usualmente copian datos de salida desde el espacio de usuario en el espacio de kernel antes de enviarlos al dispositivo. Entre otras aspectos, este enfoque permite al usuario continuar modificando los datos mientras la operación de salida está en progreso. Por ejemplo, esto es lo que ocurre cuando se emite un comando de impresión desde un procesador de palabra y luego se continúan realizando más modificaciones a la página que se imprimirá, incluso antes de que la solicitud de impresión se complete. Algo similar ocurre en la dirección de entrada por otras razones, como la verificación de integridad de los datos. En los sistemas o aplicaciones donde el rendimiento I/O es enormemente crítico, algo de este copiado se puede evitar. Sólo si el usuario quiere hacer cambios a los datos antes de que la impresión esté completa, el sistema operativo hará una copia para imprimir.

El rendimiento total I/O, o el número de operaciones I/O realizadas por unidad de tiempo, es de interés cuando cada operación I/O es bastante simple, pero muchas de tales operaciones se deben realizar en rápida sucesión. En la mayoría de las aplicaciones, la latencia también es de preocupación (por ejemplo, en una red de cajeros automáticos de banco ligados a una instalación de cómputo central), ello conduce a desafiantes problemas de calendarización. Esta situación se suscita debido a que un rendimiento total elevado dicta el uso de colas para uso óptimo de los recursos, mientras que la latencia baja señala que cada solicitud se atienda tan pronto como sea posible.

Observe que muchos de los problemas y desafíos descritos aquí surgen de los deseos por ocultar las complejidades de I/O del usuario y para hacer que los programas de aplicación y sistema sean en mayor medida independientes de la tecnología de dispositivo I/O. Estas metas se lograron al introducir capas de software que protegen a los usuarios de características de dispositivo hardware. Remover algunas de estas capas de complejidad constituye una forma de mejorar la latencia y el rendimiento total I/O. Lo anterior se hace rutinariamente, por ejemplo, en consolas de juego, donde el alto rendimiento total requerido para salida gráfica sofisticada necesita interacción directa entre el hardware gráfico y la unidad de pantalla con cabecera mínima. De hecho, los diseñadores de tales consolas de juego han sido tan exitosos en afinar sus procesamientos gráficos para máximo rendimiento, que la idea de usar un gran número de tales consolas para construir una supercomputadora es el objetivo de más de un equipo de investigación.

Para lograr la misma I/O de cabecera baja para sistemas de propósito general, se deben reducir tanto el número de capas involucradas en realizar I/O como la latencia en cada capa. Lo último en reducción del número de capas consiste en proporcionar a los usuarios control directo de los dispositivos I/O para

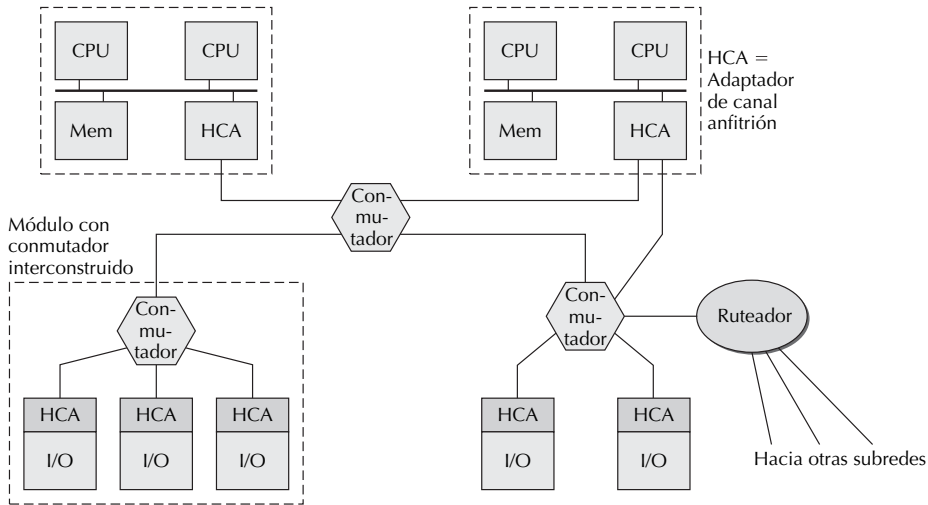


Figura 22.6 Ejemplo de configuración para la I/O Infiniband distribuida.

ciertas actividades I/O críticas. Para reducir la latencia en cada capa, se pueden buscar tanto la especialización de las rutinas para evitar operaciones que no son aplicables a un dispositivo I/O particular, como la reducción en la cantidad de copiado de datos en la medida posible. Al nivel de hardware, proporcionar a los dispositivos más interfaces inteligentes y la habilidad para comunicarse mediante una red basada en conmutadores, en oposición a varias capas de buses y los adaptadores asociados, es el impulso del nuevo estándar I/O Infiniband. Como se muestra en la figura 22.6, este esquema permite que los dispositivos I/O se comuniquen uno con otro directamente sin colocar una carga sobre el CPU o incluso sobre sus buses.

PROBLEMAS

22.1 Entrada/salida a través de ciclo de espera

- ¿El fragmento de programa del ejemplo 22.2 funcionará correctamente cuando `beq` sea una bifurcación retardada con una rendija de retardo? Si no resulta así, modifique la secuencia de instrucciones para que funcione. Para el caso de que funcione así, explique por qué.
- Repita la parte a) para el ejemplo 22.3.

22.2 Entrada y eco de datos en línea de comando

Convierta las secuencias de instrucciones de los ejemplos 22.2 y 22.3 en procedimientos MiniMIPS. Luego escriba una secuencia de instrucciones MiniMIPS que cause que los caracteres ingresados en el teclado se salven en memoria y también se presenten en la pantalla, hasta que la tecla Enter se presione. En este punto, el símbolo *null* se anexa al final de la cadena almacenada y la cadena que termina en *null* pasa a un procedimiento para análisis y acción adecuada.

22.3 Repetición en un teclado

En la mayoría de las computadoras, si usted oprime una tecla en el teclado y la sostiene, el carácter se repetirá. Por tanto, retener la tecla de guión o de guión bajo creará una línea punteada o una línea sólida, respectivamente. La tasa de repetición está determinada por un parámetro definido por usuario. Discuta cómo se puede manejar este aspecto de entrada a datos de teclado en el contexto del ejemplo 22.2.

22.4 Registros de estatus de dispositivo

En los registros de control (estatus) de dispositivo de la figura 22.1:

- Sugiera posibles usos para al menos otros dos bits en el registro de control de teclado (además de R e IE).
- Repita la parte a) para el registro de control de pantalla.
- Repita la parte a) para un tercer dispositivo I/O que elija y describa.

22.5 Direccionamiento lógico para dispositivos I/O

En el esquema I/O que se muestra en la figura 22.2:

- ¿Qué ocurre si dos dispositivos de entrada diferentes se asignan a la misma dirección de dispositivo?
- Repita la parte a) para dispositivos de salida.
- ¿Es posible conectar dos o más teclados diferentes al bus? ¿Puede pensar en una situación en la que esta capacidad pueda ser útil?
- Repita la parte c) para unidades de pantalla.

22.6 Sondeo de un teclado con *buffer*

De acuerdo con la suposición de que dos presiones de tecla sucesivas en un teclado nunca pueden estar separadas menos de 0.1 s, una tasa de sondeo de 10/s, como en el ejemplo 22.4, es adecuada. Suponga que, además, se sabe que no más de diez presiones de tecla pueden ocurrir en cualquier intervalo de 2 s. Al proporcionar un *buffer* en el teclado, se puede reducir la tasa de sondeo mientras todavía se capturan todas las presiones de tecla.

- Determine el tamaño y organización adecuados del *buffer* de teclado.
- Proponga una realización en hardware del *buffer* de teclado.
- ¿Cuál es la tasa de sondeo adecuada con el *buffer* de la parte a)?
- ¿Cómo puede asegurar el CPU que la tasa de sondeo de la parte c) se cumple?
- ¿Bajo qué condiciones un *buffer* mayor ayudaría a reducir la tasa de sondeo incluso todavía más?

22.7 Sondeo de una unidad de disco duro

Considere el sondeo de una unidad de disco, con las características de disco y CPU dadas en el ejemplo 22.6.

- Determine la frecuencia de reloj mínima para el CPU si debe mantener el disco duro.
- ¿Proporcionar un *buffer* para la unidad de disco, como se hizo para el teclado del problema 22.6, permitiría al CPU sondear menos frecuentemente? Justifique su respuesta.
- ¿Qué cambios a las suposiciones permitirían lograr el sondeo con un CPU de 500 MHz?
- ¿Un CPU de 2 GHz sería capaz de sondear dos discos duros? ¿Y cuatro discos?

22.8 I/O de disco a través de interrupciones

En concordancia con el ejemplo 22.7, y con las suposiciones que ahí se mencionan, una unidad de disco utiliza menos de 5% del tiempo de CPU si se usan interrupciones para salida y entrada.

- Explique por qué este resultado no implica que un CPU puede manejar I/O desde o hacia 20 discos.
- Explique por qué incluso dos discos duros pueden no ser manejados adecuadamente a menos que se implementen algunas provisiones especiales que subraye.

22.9 Transferencias de datos DMA

En la sección 22.5 se observó que los controladores DMA con la habilidad para transferir datos en explosiones más cortas, mientras regresa el control de bus al CPU después de cada transferencia parcial, ofrece el beneficio de que el CPU nunca tenga que esperar mucho para usar el bus.

- ¿Puede pensar en alguna desventaja para este esquema?
- ¿Qué factores influyen la elección de la longitud de explosión para transferencia parcial? ¿Cuáles son las negociaciones?
- Repita la parte b) para el caso cuando dos controladores DMA se unen al mismo bus.

22.10 Efectos de DMA sobre rendimiento de CPU

- La operación de un canal DMA, y sus efectos sobre el rendimiento CPU, se puede modelar en la siguiente forma. El tiempo se mide en términos de ciclos de bus y todas las cabeceras, incluido el retardo de arbitraje, se ignoran. Cuando el canal DMA no usa el bus durante cierto ciclo de bus, se puede volver activo con probabilidad 0.1 en el ciclo de bus subsecuente. Hasta la activación, obtiene y usa el bus durante 12 ciclos antes de abandonarlo. ¿Cuál es la probabilidad a largo plazo de que el CPU pueda usar el bus en algún ciclo específico? ¿Cuál es el tiempo de espera esperado para el CPU?
- Repita la parte a), y esta vez suponga que el DMA libera el bus durante un ciclo después de cada cuatro ciclos de uso; por tanto, los 12 ciclos de uso ocurren en tres plazos, separados por dos ciclos libres.

22.11 Ancho de banda I/O efectivo

Este problema es continuación del ejemplo 22.9.

- Grafique la variación en rendimiento total contra tamaño de bloque (figura 22.5), con el uso de escala logarítmica para el tamaño de bloque en el eje horizontal.
- Repita la parte *a*), esta vez con escalas logarítmicas en ambos ejes.
- ¿Las gráficas de las partes *a*) y *b*) proporcionan alguna información adicional acerca de la elección del tamaño de bloque? Discuta.

22.12 Ancho de banda I/O efectivo

El ejemplo 22.9 es un poco irreal en cuanto a que asocia la latencia de acceso promedio de 10 ms con cada acceso a disco. En la realidad, si la lectura de bloques de datos más grandes, tienen algún sentido, los accesos a bloques más pequeños deben involucrar mayor cantidad de localidad. Asimismo, un acceso verdaderamente aleatorio quizá tendrá una mayor latencia que el promedio de 10 ms medido sobre todos los accesos. Discuta cómo se pueden explicar estas consideraciones en un análisis más realista. Establezca todas sus suposiciones claramente y cuantifique el efecto del tamaño de bloque sobre el rendimiento total I/O efectivo, como en la figura 22.5.

22.13 I/O Infiniband distribuida

Prepare un informe de cinco páginas acerca del esquema I/O Infiniband distribuida, con enfoque particular sobre la estructura del adaptador de canal anfitrión y los elementos de conmutación (que se muestran en la figura 22.6). Ponga especial atención a cómo difieren los dos elementos en estructura y función.

22.14 Rendimiento I/O

Una aplicación bancaria representa un ejemplo de una carga de trabajo de procesamiento de transacción. Enfóquese sólo en las transacciones de cajeros automáticos realizadas por clientes del banco e ignore todas las otras transacciones que inicie el personal del banco (por ejemplo, procesamiento de cheques, apertura de cuentas, actualizaciones varias). Si toma una visión simplificada, cada transacción en cajero automático involucra tres accesos a disco (autenticación, lectura de datos de cuenta y actualización de cuenta).

- Para cada una de las memorias de disco mencionadas en la tabla 19.1, estime el número máximo de transacciones que se pueden soportar por segundo (ignore la inconveniencia de discos muy pequeños para este tipo de aplicación).
- Si se permite 50% de rendimiento total de repuesto para expansión futura, y 40% de desperdicio de ancho de banda de disco debida a cabecera administrativa, ¿cuántas de cada unidad de disco consideradas en la parte *a*) se necesitarían para soportar un rendimiento total de 200 transacciones por segundo?
- ¿Cuáles son algunas de las razones para una transacción bancaria real en cajero automático que requiere más de tres accesos a disco?
- ¿Cómo afectan el rendimiento total de procesamiento de transacción los accesos adicionales en la parte *c*)?

22.15 Rendimiento I/O

Una computadora usa un bus I/O de 1 GB/s que puede alojar hasta diez controladores de disco, cada uno capaz de conectarse a diez unidades de disco. Suponga que la memoria principal interpolada no es un problema, en el sentido de que puede suministrar o aceptar datos a la tasa pico del bus I/O. Cada controlador de disco añade una latencia de 0.5 ms a una operación I/O y tiene una tasa de datos máxima de 200 MB/s. Considere el uso de las memorias de disco citadas en la tabla 19.1 e ignore lo improbable de que se usarían discos muy pequeños para grandes capacidades de almacenamiento. Calcule la tasa I/O máxima que puede lograr en este sistema con cada tipo de disco. Suponga que cada operación I/O involucra un solo sector elegido aleatoriamente. Establezca todas sus suposiciones con claridad.

22.16 Teoría de colas

Un controlador de disco que recibe solicitudes de acceso a disco se puede modelar mediante una cola que contenga las solicitudes y un servidor que satisfaga las solicitudes en orden FIFO. Cada solicitud entrante pasa algún tiempo en la cola y requiere cierto tiempo de servicio para el acceso real. Suponga que el disco puede realizar un promedio de un acceso cada 10 ms. En este orden de ideas, se supone que el tiempo de servicio siempre es $T_{\text{servicio}} = 10$ ms. Sea u la utilización de disco; por ejemplo, $u = 0.5$ si el disco realiza 50 accesos por segundo ($50 \times 10 \text{ ms} = 500 \text{ ms}$). Entonces, si supone

llegadas aleatorias de solicitud con una distribución de Poisson, la teoría de colas dice que el tiempo que una solicitud de acceso pasa esperando en la cola es $T_{\text{cola}} = 10u/(1 - u)$ ms, donde el factor 10 es el tiempo de servicio en milisegundos. Por ejemplo, si $u = 0.5$, cada solicitud pasa un promedio de 10 ms en la cola y 10 ms para el acceso real, para una latencia promedio total de 20 ms.

- a) ¿Cuál es la utilización del disco si, en promedio, se emiten 80 solicitudes I/O a disco por segundo?
- b) Con base en la respuesta a la parte a), calcule la latencia total promedio para cada solicitud de acceso a disco.
- c) Repita la parte a), pero suponga que el disco se sustituye con un disco más rápido con latencia promedio de 5 ms.
- d) Repita la parte b) con base en la respuesta a la parte c).
- e) Calcule la aceleración de la parte d) sobre la parte b) (es decir, el impacto sobre el rendimiento del disco más rápido).

el tiempo de respuesta promedio. Por ejemplo, si los clientes de un banco llegan, en promedio, uno cada dos minutos (tasa de llegada de 0.5/min) y un cliente pasa un promedio de diez minutos en el banco, entonces en promedio habrá $0.5 \times 10 =$ cinco clientes adentro del banco. Con base en la explicación del problema 22.16, y si supone un solo empleado en el banco, un tiempo de servicio s para cada cliente implica $T_{\text{cola}} = su/(1 - u)$, donde u , la utilización del empleado, es $0.5s$ (es decir, el producto de la tasa de llegada y el tiempo de servicio a cliente).

- a) Con base en dicha información, ¿cuál es el tiempo de servicio para cada cliente?
- b) ¿Cuál es la longitud esperada de la cola (número de clientes esperando) en el banco?
- c) Relacione este ejemplo del banco con la entrada/salida del disco.
- d) Discuta el impacto de tener un empleado que trabaja el doble de rápido que el empleado solo original contra tener dos empleados con la misma velocidad de trabajo. Observe que el cálculo exacto de los impactos de la segunda opción es complicado; de este modo, presente un argumento intuitivo para los méritos relativos de cada opción.
- e) ¿Cuál sería la contraparte I/O del disco de tener más de un empleado en el banco?

22.17 Teoría de colas

La ley de Little en teoría de colas establece que bajo condiciones de equilibrio el número promedio de tareas en un sistema es el producto de su tasa de llegada y

REFERENCIAS Y LECTURAS SUGERIDAS

- | | |
|----------|--|
| [Brya03] | Bryant, R. E. y D. O'Hallaron, <i>Computer Systems: A Programmer's Perspective</i> , Prentice Hall, 2003. |
| [Henn03] | Hennessy, J. L. y D. A. Patterson, <i>Computer Architecture: A Quantitative Approach</i> , Morgan Kaufmann, 3a. ed., 2003. |
| [Patt98] | Patterson, D. A. y J. L. Hennessy, <i>Computer Organization and Design: The Hardware/Software Interface</i> , Morgan Kaufmann, 2a. ed., 1998. |
| [Rash88] | Rashid, R., A. Tevanian Jr, M. Young, D. Golub, R. Baron, D. Black, W. J. Bolosky y J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", <i>IEEE Trans. Computers</i> , vol. 37, núm. 8, pp. 896-908, agosto de 1988. |
| [Sanc03] | Sanchez, J., y M. P. Canton, <i>The PC Graphics Handbook</i> , CRC Press, 2003. Cap.18, "Keyboard and Mouse Programming", pp. 509-532. |
| [Scha01] | Schaelicke, L., <i>Architectural Support for User-Level Input/Output</i> , PhD dissertation, University of Utah, diciembre de 2001. http://www.cse.nd.edu/~lambert/pdf/dissertation.pdf |
| [Silb02] | Silberschatz, A., P. B. Galvin y G. Gagne, <i>Operating Systems Concepts</i> , Wiley, 6a., 2002. |
| [Wolf01] | Wolf, W., <i>Computers as Components</i> , Morgan Kaufmann, 2001. |

■ CAPÍTULO 23

BUSES, LIGAS E INTERFASES

“Nunca subestimes el ancho de banda de una camioneta llena de cintas que se lanza por la carretera.”

Andrew Tannenbaum

“Computadoras digitales: personas que cuentan con sus dedos.”

De El libro de frases en “B.C”, tira cómica de Hart

TEMAS DEL CAPÍTULO

- 23.1** Ligas intra e intersistema
- 23.2** Buses y su presentación
- 23.3** Protocolos de comunicación de bus
- 23.4** Arbitraje y rendimiento de bus
- 23.5** Fundamentos de interfases
- 23.6** Estándares en la creación de interfases

Es común usar ligas (links, vínculos, enlaces) compartidas o buses para interconectar muchos subsistemas. Esto no sólo conduce a menos alambres y pines, sino que mejora la efectividad de costo y la confiabilidad, también contribuye a la flexibilidad y expandibilidad. En este capítulo se revisan tales conexiones compartidas y dos temas clave en su diseño y uso: el arbitraje, para garantizar que sólo un emisor ponga datos en el bus en un momento dado; y la sincronización, para ayudar a verificar que el receptor ve los mismos valores de datos que transmitió el emisor. La creación de interfases de los dispositivos periféricos a las computadoras involucra temas complejos de detección de señal y reconstrucción. Por esta razón se han desarrollado muchas interfases estándar que definen completamente los parámetros de señal y protocolos de intercambio. Un dispositivo periférico que produce o recibe señales en concordancia con uno de muchos protocolos de bus/link estándar se puede conectar en una computadora y usarse con mínimo esfuerzo.

■ 23.1 Ligas intra e intersistema

La conectividad intrasistema se establece principalmente mediante delgadas capas metálicas depositadas en, y separadas por, capas de aislador sobre microchips o placas de circuito impreso. Tales ligas son o punto a punto, que conectan una terminal de un subsistema particular a una terminal de otro, o buses compartidos que permiten que múltiples unidades se comuniquen a través de él. Dada la

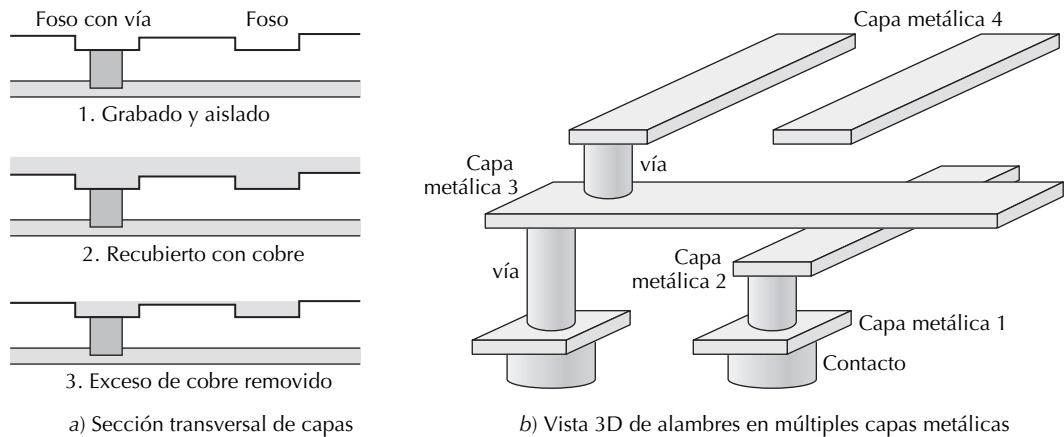


Figura 23.1 Múltiples capas metálicas proporcionan conectividad intrasistema sobre microchips o placas de circuito impreso.

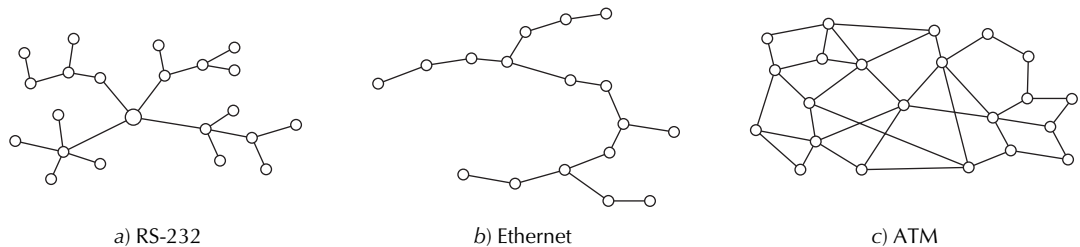


Figura 23.2 Ejemplo de esquemas de conectividad intersistema

importancia de los buses en la implementación y rendimiento del sistema, se tratan por separado en las secciones 23.2 a 23.4. Los actuales procesos de fabricación electrónica permiten múltiples capas metálicas a nivel de chip y placa, lo anterior permite conectividad bastante compleja (figura 23.1). Los alambres se “depositan” capa a capa, así como el resto de los pasos de procesamiento. Por ejemplo, para depositar dos alambres paralelos en una capa, se graban dos fosos donde deben aparecer aquéllos (figura 23.1a). Después de que se recubre la superficie con una delgada capa de aislador, se deposita una capa de cobre que llena los fosos y cubre el resto de la superficie. El pulido de la superficie remueve el cobre excesivo, excepto en los fosos donde se formaron los alambres.

Cuando los componentes deben ser removibles e intercambiables, se ligan al resto del sistema mediante *sockets* (clavijas) en los que se pueden conectar o cables especiales con conectores estándar en cualquier extremo. Algunos de los estándares relevantes a este respecto se discuten en la sección 23.6.

Más allá de la frontera de la placa de circuito, se usan para comunicación alambres, cables y fibras ópticas, dependiendo del costo y los requisitos de ancho de banda. La figura 23.2 muestra algunas de estas opciones. La conectividad intersistema toma una diversidad de formas, que van desde alambres cortos que ligan unidades cercanas a redes de área local y ancha. Las estructuras de interconexión requeridas se caracterizan por:

■ TABLA 23.1 Resumen de tres esquemas de interconexión

Propiedades de interconexión	RS-232	Ethernet	ATM
Máxima longitud de segmento, m	Decenas	Cientos	1 000
Máxima amplitud de red, m	Decenas	Cientos	Ilimitada
Tasa de bits, Mb/s	Hasta 0.02	10/100/1 000	155-2500
Unidad de transmisión, B	1	Cientos	53
Latencia extremo a extremo típica, ms	<1	Decenas de cientos	Cientos
Dominio de aplicación típico	Entrada/salida	LAN	Backbone
Complejidad o costo de transceptor	Baja	Baja	Alta

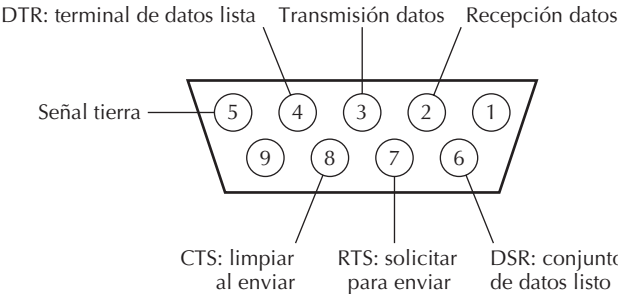


Figura 23.3 Interfaz serial RS-232 con conectores de nueve pines.

Distancia/amplitud:	unos cuantos metros a miles de kilómetros
Tasa de datos:	unos cuantos kilobits por segundo a muchos gigabits por segundo
Topología:	bus, estrella, anillo, árbol, etcétera
Compartición de línea:	punto a punto (dedicada) o multigota (compartida)

Para darse una idea acerca de las características de interconexiones para diferentes distancias y tasas de datos (*data rate*), en el resto de esta sección (figura 23.2) se describen tres ejemplos de estándares de interconectividad intersistema. La tabla 23.1 contiene un resumen de las características más importantes.

Desde sus inicios a principios de la década de 1960 como un estándar EIA (Electronics Industries Association), RS-232 ha sido un esquema ampliamente usado para conectar equipo externo como terminales a computadoras. Es un método serial en el que la transmisión ocurre bit por bit a través de una sola línea de datos, que está soportada por algunas otras líneas para varias funciones de control, incluido protocolo de intercambio (*handshaking*). En realidad existen tres alambres para soportar la transmisión de datos seriales totalmente doble: un alambre de datos para cada dirección, más una tierra (*ground*) compartida. El conector RS-232 completo tiene 25 pines, pero la mayoría de las PC usan una versión reducida de nueve pines, que se muestra en la figura 23.3. Las cuatro señales de control (CTS, RTS, DTR, DSR) permiten *handhsaking* entre transmisor y receptor de datos (sección 23.3). con RS-232, se transmiten símbolos planos de ocho bits o codificados por paridad, junto con tres bits adicionales como la cadena 0d...dp11, donde el 0 de la extrema izquierda es el bit de arranque, d representa datos, p es el bit de paridad o un bit de dato octavo, y los dos números 2 a la derecha son los bits de alto. Los símbolos transmitidos se pueden separar por bits 1 de inactividad que siguen a los dos bits de alto. Como consecuencia de su corto rango (decenas de metros) y tasa de datos relativamente baja (0.3-19.6 Kb/s), RS-232 asumió el estatus de un estándar legado en años recientes, ello significa que está soportado por cuestiones de compatibilidad con dispositivos y sistemas antiguos.

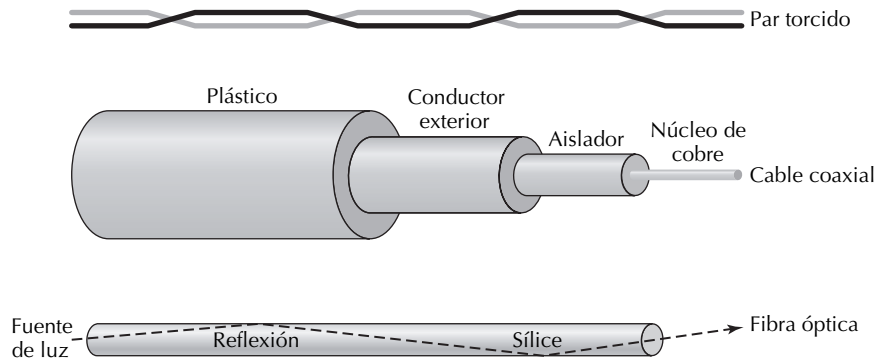


Figura 23.4 Medios de comunicación usados de manera común para conexiones intersistema.

Ethernet es el estándar de LAN (red de área local) más utilizado. En sus varias formas, Ethernet soporta tasas de transmisión de 10 Mb/s, 100 Mb/s y, recientemente, 1 Gb/s. El medio usado para transmisión puede ser un par de alambres torcidos o trenzados (*twisted*), cable coaxial o fibra óptica (figura 23.4). A Ethernet se pueden conectar múltiples dispositivos o nodos que pueden soportar una transmisión por un solo emisor en cualquier momento con una latencia del orden de 1 ms, con el uso de paquetes o marcos cuyo tamaño es de cientos de bytes. Cada nodo conectado a Ethernet tiene un transceptor (transmisor/receptor) capaz de enviar mensajes y de cuidar el tráfico de la red. Cuando ningún nodo transmite sobre el “éter” compartido, en la línea hay un voltaje fijo de +0.7 V. Esto último se conoce como la señal *sensora del portador* (*carrier sense*) o *latido* (*heartbeat*), que indica que la red está activa y disponible para su uso. Para transmitir sobre Ethernet, un nodo “escucha” un breve periodo antes de arrancar. Si dos nodos comienzan a transmitir al mismo tiempo, resulta una colisión. Cuando se detecta una colisión, los nodos participantes producen una señal “jam” (atasco) y esperan aleatoriamente durante decenas de milisegundos antes de intentarlo de nuevo. Este tipo de arbitraje de bus distribuido basado en la detección de colisiones se discutirá aún más en la sección 23.4. Un nodo inicia su transmisión al enviar una cadena de 0 y 1 conocida como preámbulo, lo anterior hace que el voltaje de la línea fluctúe entre +0.7 y -0.7 V. Si no se detecta colisión hacia el final del preámbulo, entonces el nodo envía el resto del marco. Un medio común de espitar en Ethernet es el conector RJ-45 con ocho pines que constituyen cuatro pares de líneas.

ATM (modo de transferencia asíncrono) constituye una tecnología de interconexión usada en la construcción de redes de tirada larga del tipo que se muestra en la figura 23.2c. Formada de conmutadores conectados en una topología arbitraria, una red ATM puede transmitir datos a una tasa de 155 Mb/s a 2.5 Gb/s. La transmisión no ocurre en una corriente de bit continua, sino más bien con base en enrutamiento almacenar-adelantar de paquetes de 53 B (5 B de cabecera seguidos por 48 B de datos). La diferencia con enrutamiento de paquete en Internet es que se establece una ruta específica, o conexión virtual, y todos los paquetes que comprenden una transmisión particular se adelantan a través de dicha ruta. Esto permite negociar ciertos parámetros de calidad de servicio (como la tasa de datos garantizada y la probabilidad de pérdida de paquete) al momento de establecer la conexión virtual. El contenido de los 48 B de datos de un paquete se eligió como un compromiso para equilibrar la cabecera de control en cada paquete contra los requerimientos de latencia en la comunicación por voz, donde cada milisegundo de habla se representa usualmente en ocho bytes, ello conduce a intervalos de 6 ms por paquete ATM; intervalos más largos habrían conducido a retardos audibles, así como a pausas no placenteras en el caso de pérdida de paquete. El desarrollo de ATM está gestionado por el ATM Forum [WWW], que entre sus miembros tiene muchas compañías de telecomunicaciones y varias universidades.

Las ligas intra e intersistema han asumido un papel central en el diseño de sistemas digitales. Una razón es que la densidad y el rendimiento rápidamente crecientes de los componentes electrónicos ha propiciado que el abastecimiento de la conectividad requerida a velocidades razonables sea un problema muy difícil. La propagación de señal a través de alambres se efectúa de un tercio a la mitad de la rapidez de la luz. Esto último significa que las señales electrónicas viajan:

A través el bus de memoria (10 cm)	en	1 ns
A través de una computadora grande (10 m)	en	100 ns
A través de un campus universitario (1 km)	en	10 μs
A través de una ciudad (100 km)	en	1 ms
A través de un país (2 000 km)	en	20 ms
A través del globo (20 000 km)	en	200 ms

Incluso el retardo de propagación de señal en el chip se ha vuelto no despreciable y se debe tomar en cuenta para el diseño de circuitos digitales de alto rendimiento.

■ 23.2 Buses y su presentación

Un bus representa un conector que vincula múltiples fuentes y sumideros de datos. La ventaja principal de la conectividad basada en bus es que, teóricamente, permite cualquier número de unidades para comunicar entre ellos mismos, y cada unidad requiere sólo un puerto de entrada y otro de salida. Lo anterior minimiza alambrear o cablear y permite la flexibilidad de sumar nuevas unidades con cambio o perturbación mínimos. Sin embargo, prácticamente existe un límite superior al número de unidades que pueden compartir un solo bus; dicho límite depende de la tecnología y rapidez del bus. Muchas computadoras de bajo perfil se benefician de la simplicidad y economía de un solo bus compartido que conecta a todos los componentes del sistema (figura 21.1). Los sistemas de mayor rendimiento pueden incluir múltiples buses o usar conectividad punto a punto para mayor rapidez y ancho de banda.

Un bus usualmente consta de un haz de líneas de control, un conjunto de líneas de dirección y varias líneas de datos, como se muestra en la figura 23.5. Las líneas de control portan muchas señales necesarias para controlar las transferencias de bus, los dispositivos conectados al bus y los protocolos de *handshaking* asociados. Las líneas de dirección portan información acerca de la fuente o destino de datos (localidad de memoria o dispositivo I/O). El ancho de dirección más común en uso actual es de 32 bits, pero en muchos sistemas existen direcciones más estrechas y más anchas. Las líneas de datos portan valores de datos a enviar desde una unidad hacia otra. Los buses seriales tienen una sola línea de datos, mientras que los buses paralelos portan 1-16 bytes (8-128 bits) de datos simultáneamente. Las líneas de dirección y datos se pueden compartir (multiplexadas), pero esto último conduce a un rendimiento inferior.

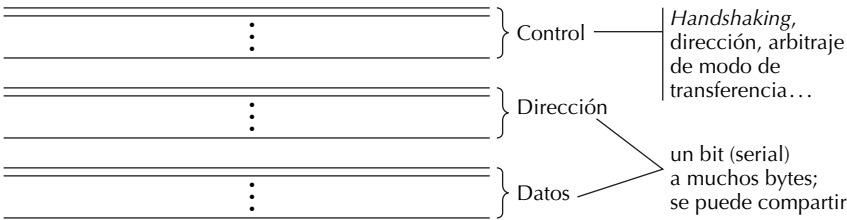


Figura 23.5 Tres conjuntos de líneas que se encuentran en un bus.

Una computadora tradicional puede usar más o menos una docena de diferentes tipos de buses. Existen tres razones principales para esta multiplicidad:

1. Los *buses de legado* son buses que existieron en el pasado y deben soportarse para que los usuarios continúen usando sus antiguos dispositivos periféricos. Los ejemplos de buses de legado incluyen PC Bus, ISA, RS-232 y el puerto paralelo.
2. Los *buses estándar* siguen estándares internacionales populares y se requieren para conectar a la computadora los dispositivos periféricos más actuales. Los ejemplos de aquéllos, que se discuten aún más en la sección 23.6, incluyen PCI, SCSI, USB y Ethernet.
3. Los *buses propietarios* están diseñados a la medida para maximizar la tasa de transferencia de datos entre componentes de sistema críticos, y, por tanto, el rendimiento global del sistema. Estos buses están diseñados para dispositivos específicos conocidos al momento de diseño y no se espera que sean compatibles con otros dispositivos o estándares.

En lo que sigue se revisan ciertas características de los buses que son comunes entre las tres categorías anteriores. Las especificaciones de bus más importantes incluyen:

Tasa de datos: Si se excluyen los antiguos buses lentos, la tasa de datos varía de muchos megabytes por segundo, a través de decenas o cientos de megabytes por segundo (por ejemplo, en Ethernet) hasta muchos gibabytes por segundo en buses de memoria rápida que, por lo general, usan reloj de 100-500 MHz con un canal de datos de 8 B de ancho.

Número máximo de dispositivos: Más dispositivos aumentan la carga del bus, así como la cabecera de arbitraje, ello hace más lento el bus. Este parámetro varía de pocos (por ejemplo, 7 para SCSI estándar) hasta muchos miles (por ejemplo, Ethernet).

Método de conexión: Esta característica tiene que ver con los tipos de conector y cable que se pueden usar para ligar al bus, así como facilitar el agregado o remoción de dispositivos. Por ejemplo, USB es *conectable directo* (*hot-pluggable*), lo que significa que los dispositivos se pueden agregar o remover aun cuando el bus esté en operación.

Fortaleza y confiabilidad: Es importante saber cuán bien tolera un bus el malfuncionamiento de un dispositivo que está conectado a él, o varias irregularidades eléctricas como cortocircuitos, que pueden ocurrir antes o durante la transmisión de datos.

Parámetros eléctricos: Éstos incluyen niveles de voltaje, rangos de corriente, protección de sobretensión y cortocircuito, intermodulación y requerimientos de energía.

Cabecera de comunicación: Los bits de control y verificación de error se suman a los datos transmitidos para control, coordinación o confiabilidad mejorada. Esta cabecera tiende a ser lenta para buses rápidos cortos y muy significativa para transmisiones de tirada larga.

Los buses también se pueden clasificar de acuerdo con el esquema de control o arbitraje. En el esquema de control más simple hay un solo bus maestro que inicia todas las transferencias de bus (el maestro envía o recibe datos). Por ejemplo, el CPU puede ser el maestro que envía datos hacia o recibir datos desde cualquier unidad conectada al bus. Esto es muy simple de implementar, pero tiene el doble inconveniente de desperdiciar tiempo de CPU y usar dos transferencias de bus para enviar una palabra de datos desde disco a memoria.

Ligeramente más flexible es un esquema en el que el bus maestro puede delegar al control del bus a otros dispositivos durante un número específico de ciclos o hasta que se señale la conclusión de la transferencia. El esquema DMA de la sección 22.5 es un ejemplo privilegiado de este método.

Los buses con maestros múltiples pueden tener un árbitro que decida cuál maestro obtiene el uso del bus. De manera alterna, puede usarse un esquema de control distribuido por medio del cual se detecten y lidien con “colisiones” a través de la retransmisión en un momento posterior (por ejemplo, en Ethernet). El método usado para arbitraje de bus tiene un profundo efecto sobre el rendimiento. Los métodos de arbitraje y rendimiento de bus se discuten en la sección 23.4.

Ejemplo 23.1: Tomar en cuenta la cabecera de arbitraje de bus Un bus de 100 MHz tiene 8 B de ancho. Suponga que 40% del ancho de banda del bus se desperdicia debido a arbitraje y otra cabecera de control. ¿Cuál es el ancho de banda neto del bus disponible a otros dispositivos I/O y al CPU después de tomar en cuenta las necesidades de una unidad de pantalla de video de 1024×768 pixeles con 4 B/pixel y una tasa de regeneración de 70 Hz?

Solución: La tasa de datos de la unidad de pantalla es $1024 \times 768 \times 4 \times 70 \cong 220$ MB/s. El ancho de banda total efectivo del bus es $100 \times 8 \times 0.6 = 480$ MB/s. El ancho de banda neto es, por tanto, $480 - 220 = 260$ MB/s, que servirá al CPU y a los restantes dispositivos I/O. Observe que se supuso que la memoria puede suministrar datos a la tasa de bus efectiva de 480 MB/s. Esto no es irracional; representa 48 bytes cada 100 ns.

23.3 Protocolos de comunicación de bus

Los buses se dividen en dos clases amplias de síncronos y asíncronos. Dentro de cada clase, los buses se pueden diseñar a la medida o basarse en el estándar industrial prevaleciente. En un *bus síncrono*, una señal de reloj es parte del bus y los eventos ocurren en ciclos de reloj específicos, de acuerdo con un calendario acordado (*protocolo de bus*). Por ejemplo, se puede colocar una dirección de memoria en el bus en un ciclo, donde la memoria requerida responda con los datos en el quinto ciclo de reloj. La figura 23.6 contiene una representación del protocolo precedente. Los buses síncronos son adecuados para un número pequeño de dispositivos de igual o comparable rapidez que se comunican a través de distancias cortas (de otro modo, la carga de bus y el sesgo de reloj se vuelven problemáticos).

Los buses asíncronos pueden acomodar dispositivos de varias rapideces que se comuniquen a través de distancias más largas, pues la temporización fija del bus síncrono se sustituye con un protocolo de *handshaking*. La secuencia típica de eventos en un bus asíncrono, para el caso de entrada de datos o lectura de acceso a memoria (figura 23.7) se describe a continuación. La unidad que solicita entrada

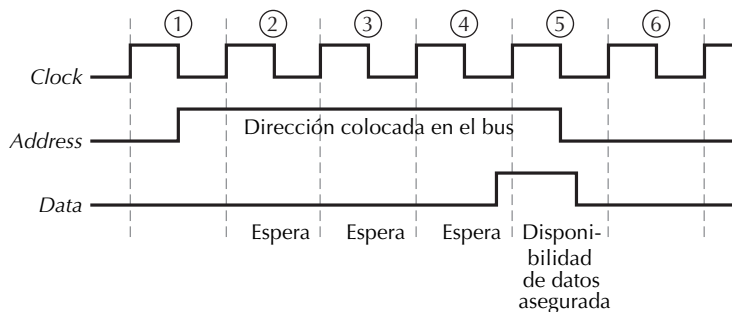


Figura 23.6 Bus síncrono con dispositivos de latencia fija.

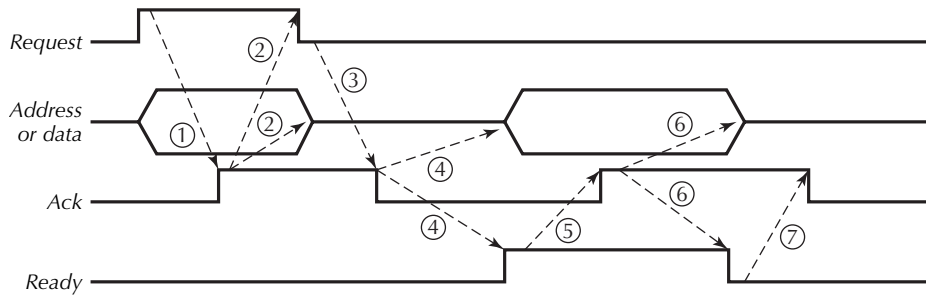


Figura 23.7 *Handshaking* en un bus asíncrono para una operación de entrada (por ejemplo, leer desde memoria).

postula la señal *Request*, mientras al mismo tiempo pone la información de dirección en un bus de dirección. El destinatario anota ① en esta solicitud, reconoce la solicitud (postula la señal *Ack*) y lee la información de dirección del bus. A su vez, el solicitante en turno anota ② del reconocimiento, despostula la señal *Request* y remueve la información del bus. A continuación, el destinatario anota ③ la despostulación de la señal *Request* y despostula su reconocimiento. Cuando los datos solicitados están disponibles ④, la señal *Ready* se postula y los datos se colocan en el bus. Ahora el solicitante anota ⑤ la señal *Ready*, postula la señal *Ack* y lee los datos del bus. El último reconocimiento ⑥ conduce a que el emisor despostule la señal *Ready* y remueva los datos del bus. Finalmente, el solicitante, al anotar ⑦ la despostulación de la señal *Ready*, despostula la señal *Ack* para terminar la transacción de bus.

Para hacer interfaz de señales asíncronas con circuitos síncronos (cronometrados), es necesario usar sincronizadores; éstos son circuitos que fuerzan ciertas restricciones de temporización relacionadas con transiciones de señales necesarias para la operación correcta de los circuitos síncronos (sección 2.6). Los sincronizadores no sólo agregan complejidad a los circuitos digitales, también introducen retardos que nulfican algunas de las ventajas de la operación asíncrona.

Dentro del sistema de una computadora, el bus procesador-memoria usualmente está diseñado a la medida y es síncrono. El bus de placa madre (*backplane*) puede ser estándar o a la medida. Finalmente, el bus I/O usualmente es estándar y asíncrono. El bus PCI se revisa como ejemplo en el resto de esta sección. En la sección 23.6 se describen algunas otras interfases estándar usadas comúnmente.

El bus estándar PCI (interconexión de componentes periféricos), desarrollado por Intel a principios de la década de 1990, ahora está gestionado por PCI-SIG [WWW], un grupo de interés especial con cientos de compañías miembros. Aun cuando PCI se desarrolló como un bus I/O, comparte algunas de las características de un bus de alto rendimiento, ello permite que se le use como punto focal por todas las comunicaciones intersistema. Por ejemplo, PCI combina operación síncrona (cronometrada) con la provisión de inserción de ciclos de espera por dispositivos más lentos que no pueden competir con la rapidez del bus. La siguiente descripción se basa en el estándar original ampliamente usado, que ahora se aumentó en varias formas, incluidas PCI-X, mini-PCI y PCI conectable directo para rendimiento mejorado, mayor conveniencia y dominios de aplicación extendidos. La descripción también refleja los requerimientos mínimos en términos de ancho de datos (32 bits) y señales de control. El ancho de datos se duplica y el número de señales de control aumenta, en la versión opcional de 64 bits.

Para reducir el número de pines, PCI tiene 32 líneas *AD* que portan dirección o datos. Hay cuatro líneas de comando activas bajas, *C/BE'*, que especifican la función a realizar y también duplican como señales habilitar byte durante la transmisión real de datos. Los comandos típicos incluyen leer memo-

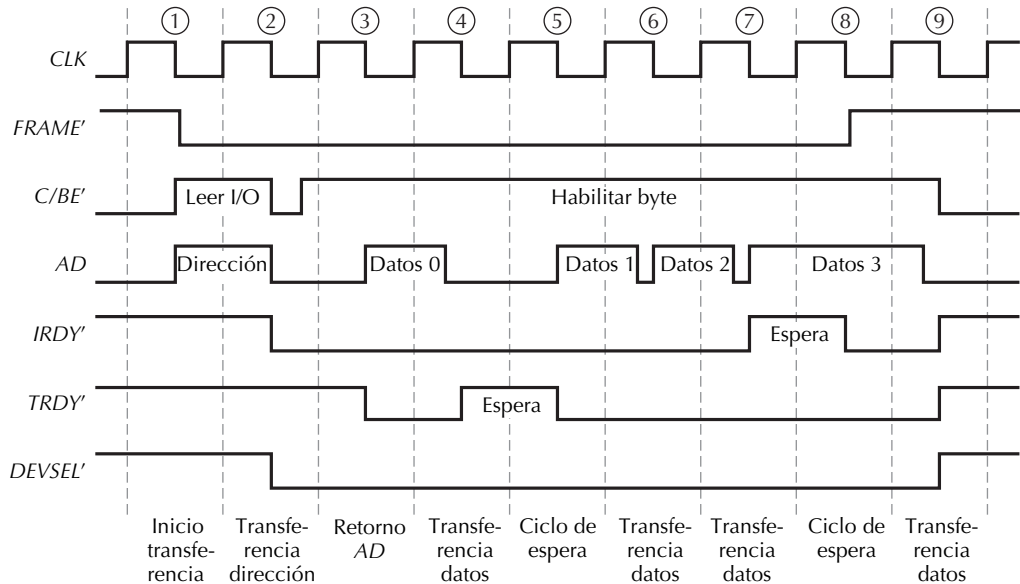


Figura 23.8 Operación leer I/O a través de bus PCI.

ria, escribir memoria, leer I/O, escribir I/O y reconocimiento de interrupción. La función habilitar byte (*byte-enable*) indica cuál de los cuatro bytes portan datos válidos. Existen seis señales de control de interfaz básicas que incluyen:

<i>FRAME'</i>	Permanece activa durante la duración de una transacción, ello la delimita
<i>IRDY'</i>	Señala que el iniciador está listo
<i>TRDY'</i>	Señala que el blanco está listo
<i>STOP'</i>	Solicita que el maestro detenga la transacción actual

Adicionalmente, tres pines se dedica a *CLK* (reloj), *RST'* (reset) y *PAR* (paridad), dos bits se usan para notificar error y dos bits se dedican para arbitraje (*REQ'* y *GNT'*, aplicable sólo a la unidad maestra).

La figura 23.8 muestra una transferencia de datos típica sobre el PCI. La transacción de bus comienza en ciclo de reloj ① cuando el bus maestro coloca una dirección y un comando (leer I/O) en el bus; también postula *FRAME'* para señalar el comienzo de la transacción. La transferencia de dirección actual desde la unidad iniciadora hacia la unidad blanco ocurre en ciclo de reloj ② cuando todas las unidades unidas al bus examinan las líneas de dirección y comando para determinar si una transacción está direccionada hacia ellas; en consecuencia, la unidad seleccionada se prepara para participar en el resto de la transacción de bus. El siguiente ciclo de reloj ③ representa un ciclo “de retorno” durante el cual el maestro delega las líneas *AD* y coloca un comando “habilitar byte” sobre las líneas *C/BE'* en preparación para la transmisión de datos en cada ciclo sucesivo. Iniciando el ciclo de reloj ④, una palabra de datos de 32 bit puede transmitirse en cada tic del reloj, con la tasa de transferencia real dictada por el emisor y el receptor a través de las líneas *IRDY'* y *TRDY'*. Por ejemplo, la figura 23.8 muestra dos ciclos de espera insertados en la corriente de datos: en el ciclo ⑤, el emisor (destino) no está listo para transmitir, mientras que en el ciclo ⑧, el ciclo de espera lo inserta el iniciador porque, por cualquier razón, no está preparado para recibir.

Ejemplo 23.2: Ancho de banda efectivo de bus PCI Si la transacción de bus que se muestra en la figura 23.7 es típica, ¿qué puede decir acerca del ancho de banda efectivo de un bus PCI que opera a 66 MHz?

Solución: Al tomar en cuenta otro ciclo de retorno (de los datos a la dirección) al final de la transacción que se muestra en la figura 23.7, se toman diez ciclos de reloj para cuatro transferencias de datos (16 B). Esto último representa una tasa de datos efectiva de $66 \times 16/10 = 105.6$ MB/s. La tasa de transferencia máxima teóricamente es $66 \times 4 = 264$ MB/s. Por tanto, la cabecera esperada debida a *handshaking* y ciclos de espera es 60%.

23.4 Arbitraje y rendimiento de bus

Cualquier recurso compartido requiere un protocolo que gobierne el acceso justo a los recursos por las entidades que necesiten usarlos; los buses no son la excepción. Al hecho de decidir cuál dispositivo obtiene el uso de un bus compartido y cuándo, se le conoce como *arbitraje de bus*. Un enfoque es usar un árbitro centralizado que reciba señales de solicitud de bus de un número de dispositivos y responde al postular una señal de garantía al dispositivo individual que puede usar el bus. Si los dispositivos solicitantes no están sincronizados con una señal de reloj común, las entradas al árbitro deben pasar a través de sincronizadores (figura 23.9). La garantía de bus puede aplicarse a un número fijo de ciclos (después de lo cual se supone que el bus está libre), o puede estarlo durante un periodo indefinido, en cuyo caso el dispositivo debe liberar explícitamente el bus antes de que se pueda emitir otra garantía.

Para dispositivos con asignaciones de *prioridad fija*, el árbitro de bus simplemente representa un codificador de prioridad que postula la señal de garantía asociada con el solicitante de mayor prioridad si el bus está disponible. Con prioridades fijas, el árbitro de bus rápidamente garantiza el bus a una unidad solicitante de alta prioridad, pero tiene la desventaja de potencialmente “matar de hambre” las unidades de prioridad inferior cuando las unidades de alta prioridad usen el bus intensamente. En este contexto, un esquema de *prioridad rotatoria* proporciona tratamiento uniforme a todos los dispositivos. Se pueden vislumbrar algunos esquemas intermedios que permitan prioridad mayor para algunos dispositivos que requieran atención más rápida y evite la inanición de los dispositivos de prioridad más baja. Algunas de tales alternativas se exploran en los problemas de fin de capítulo.

Un árbitro central puede volverse muy complejo cuando están involucradas gran cantidad de unidades; también limita la expandibilidad. Un esquema de arbitraje simple que se puede usar solo o en combinación con un árbitro basado en prioridad (aunque sea uno que tenga pocas entradas y salidas) es el *encadenamiento en margarita* (*daisy chaining*). Una cadena en margarita constituye un conjunto ordenado de dispositivos que comparten una sola señal de solicitud y garantía. Las señales de solicitud de todos los dispositivos en la cadena se unen y forman una sola entrada de solicitud a un árbitro centralizado (figura 23.10). En este sentido, a la cadena como un todo se le puede garantizar el permi-

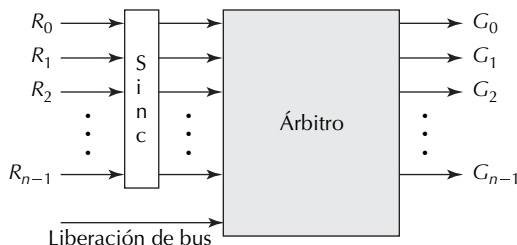


Figura 23.9 Estructura general de un árbitro de bus centralizado.

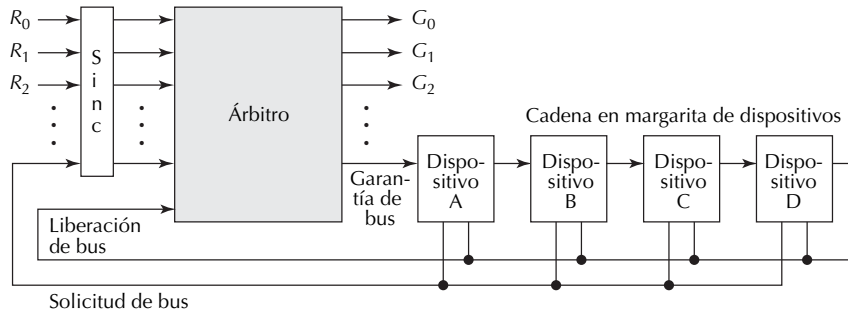


Figura 23.10 El encadenamiento en margarita permite que un pequeño árbitro centralizado atienda a un gran número de dispositivos que usan un recurso compartido.

so de usar el bus. La señal de garantía va hacia el primer dispositivo en la cadena, que puede usar el bus o adelantar la señal de garantía al siguiente dispositivo en la cadena. De esta forma, las unidades ubicadas cerca del fin de la cadena tienen menos probabilidad de obtener permiso para usar el bus; en consecuencia, no deben ser dispositivos rápidos o de alto rendimiento total.

Los esquemas de arbitraje distribuidos son más deseables porque evitan un árbitro centralizado que pueda convertirse en un problema de rendimiento. El arbitraje distribuido se puede implementar a través de *autoselección* o *detección de colisión*. En un esquema de autoselección, cada dispositivo que solicita el bus coloca un código que representa su prioridad en el bus e inmediatamente lo examina para imaginar si tiene la mayor prioridad entre todas las unidades que solicitan el bus. Desde luego, los códigos asignados a varias unidades deben ser tales que la composición de todos los códigos (OR lógica) sea adecuada para decir si una unidad solicitante tiene la mayor prioridad. Esto último tiende a limitar el número de unidades que pueden participar en el arbitraje. Por ejemplo, con un bus de ocho bits, los códigos 00000001, 00000011, 00000111, . . . , 11111111 se pueden asignar a ocho dispositivos conectados a un bus de ocho bits. Cuando muchos de estos códigos se operan OR en conjunto, resulta el código con el número más grande de 1; entonces el dispositivo que pone este código particular en el bus se selecciona a sí mismo para usar el bus.

La detección de colisión se basa en todas las unidades que solicitan usar el bus, pero vigilando el bus para ver si múltiples transmisiones se han mezclado, o chocado, unas con otras. Si ocurre una colisión, los emisores consideran sus transmisiones como fallidas e intentan más tarde, en espera de una longitud aleatoria de tiempo para minimizar las oportunidades de más colisiones. Los pretendidos receptores también serán capaces de detectar una colisión, que les causa ignorar la transmisión confusa. Para asegurar una transmisión confusa en el evento de colisión, los dispositivos en choque pueden “atascar” el bus, como se hace en Ethernet. El periodo de espera elegido aleatoriamente hace muy improbable que los mismos dispositivos choquen muchas veces, ello desperdicia ancho de banda del bus debido a colisiones repetidas. En tanto el bus no se cargue cerca de saturación y los datos se transmitan en grandes bloques, la latencia añadida debida a colisiones permanece aceptable.

El rendimiento de bus usualmente se mide en términos de la tasa de transmisión pico sobre el bus. Para un ancho de bus específico, y suponiendo que ningún ciclo de bus se pierde en arbitraje y otras cabeceras, la tasa de transmisión de datos en bus es proporcional a su frecuencia de reloj. Por tanto, se escucha acerca de buses de 166 o 500 MHz. El rendimiento (ancho de banda) del bus se puede aumentar mediante uno o más de los siguientes:

Mayor frecuencia de reloj.

Ruta de transferencia de datos más ancha (por ejemplo, 64 bits de datos en lugar de 32).

Líneas de dirección y datos separadas (no multiplexadas).

Transferencias de bloque para menos cabeceras de arbitraje.

Control de bus de grano fino; por ejemplo, liberación del bus mientras se espera que ocurra algo (protocolo de transacción dividida).

Arbitraje de bus más rápido.

Observe que un bus de, por decir, 1 GB/s se puede implementar de muchas formas. Las opciones van desde un bus serial muy rápido hasta un bus mucho más lento con ancho de datos de 256 bits.

Ejemplo 23.3: Rendimiento de bus Un bus particular debe soportar transacciones que requieran cualquier parte desde cinco hasta 55 ns para conclusión. Si supone un ancho de datos de bus de 8 B y distribución uniforme de tiempos de atención de bus dentro del rango de 5-55 ns (que hace un promedio de 30 ns), compare las siguientes opciones de diseño de bus con relación a rendimiento.

- a) Bus síncrono con un tiempo de ciclo de 60 ns, que incluye permiso para cabecera de arbitraje
- b) Bus síncrono con tiempo de ciclo de 15 ns, donde los ciclos 1-4 se usan para transferencias, según se requiera. Suponga que la señalización de conclusión y las cabeceras de arbitraje usan hasta un ciclo de bus.
- c) Bus asíncrono que requiere una cabecera de 25 ns para arbitraje y *handshaking*.

Solución: Los cálculos siguientes muestran que la opción de la parte b) ofrece el mejor rendimiento.

- a) Transferir 8 B cada 60 ns conduce a la tasa de transferencia de datos de $8/(60 \times 10^{-9})$ B/s = 133.3 MB/s.
- b) En el rango de 5-55 ns, 20% de los casos necesitan un ciclo de bus, 30% necesitan dos ciclos de bus, 30% necesitan tres ciclos de bus y los restantes 20% requieren cuatro ciclos de bus. Cuando la cabecera de un ciclo se factoriza, en promedio se usan 3.5 ciclos de bus para una transacción, ello conduce a la tasa de transferencia de $8/(3.5 \times 15 \times 10^{-9})$ B/s = 152.4 MB/s.
- c) En este caso, la tasa de transferencia promedio es 8 B cada $30 + 25 = 55$ ns, lo cual conduce a la tasa de transferencia de datos de $8/(55 \times 10^{-9})$ B/s = 145.5 MB/s.

El rendimiento no es el único criterio de diseño para buses. La fortaleza de un bus también es importante para la confiabilidad y disponibilidad del sistema. Los factores que influyen sobre la fortaleza son:

Paridad u otro tipo de codificación para detección de error.

Habilidad para rechazar o aislar unidades que funcionan mal.

Capacidad de conexión directa para operación continua durante la (des)conexión de la unidad.

Para sistemas en los que la confiabilidad o disponibilidad es muy importante, con frecuencia se usan múltiples buses para permitir la operación continua en el evento de una falla del bus.

■ 23.5 Fundamentos de interfaces

La creación de interfaces (*interfacing*) significa conectar un dispositivo externo a una computadora. Además de realizar entrada y salida de dispositivos más o menos estándares como teclados, unidad de despliegue visual e impresora, otras razones para tal conexión pueden ser:

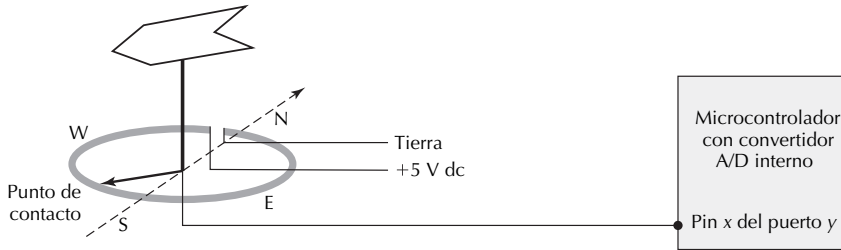


Figura 23.11 La veleta proporciona un voltaje de salida en el rango de 0-5 V, dependiendo de la dirección del viento.

Detección y recolección de datos (temperatura, movimiento, visión robótica).

Control por computadora (robótica, iluminación síntesis de voz).

Adición a las capacidades de la computadora (almacenamiento, coprocesadores, redes).

La creación de interfases puede ser tan simple como conectar un dispositivo compatible en uno de los sockets proporcionados en un bus estándar y configurar el software del sistema para permitir a la computadora comunicarse con el dispositivo a través de dicho bus. A veces, aunque más bien infrecuente en la actualidad, la creación de interfases puede involucrar hacer algo un poco más elaborado para adaptar un puerto particular de una computadora a características específicas de un dispositivo periférico.

Los microcontroladores (procesadores que están específicamente diseñados para aplicaciones de control que involucran todo, desde electrodomésticos hasta automóviles) vienen con capacidades de *interfacing* especiales, como convertidores análogo a digital (A/D) y digital a análogo (D/A), puertos de comunicación y temporizadores integrados. Los procesadores ordinarios usualmente se ponen en interfaz al conectarlos en sus buses I/O y escribir programas especiales para manejar el dispositivo agregado, con frecuencia mediante la adaptación de la rutina de manipulación de interrupción. Estos programas específicos de dispositivo se conocen como *manipuladores de dispositivo*.

Considere, como ejemplo, la detección y registro de la dirección del viento mediante una veleta que proporciona un voltaje en el rango de 0-5 V, dependiendo de la dirección. La veleta puede tener un potenciómetro interconstruido con su punto deslizante moviéndose con la dirección de viento (figura 23.11). Un voltaje de salida cercano a cero puede indicar viento al Norte, con el voltaje creciente hacia un cuarto de su máximo para el Este, la mitad para el Sur y tres cuartos para Oeste. Un microcontrolador que tiene un convertidor análogo a digital interconstruido puede aceptar el voltaje de salida de la veleta en un pin específico de uno de sus puertos. El resto se deja al software que corre en el microcontrolador. Por ejemplo, el software puede leer el contenido de un temporizador integrado que aumente cada milisegundo, verificar que el valor del temporizador indica un periodo predeterminado desde la última lectura de la dirección de la veleta, probar el pin correspondiente si es adecuado y registrar los datos en memoria.

Si se desea registrar la dirección del viento con MiniMIPS u otro procesador que no tenga una capacidad de conversión A/D interconstruida, es necesario adquirir un componente adecuado para realizar la conversión análogo a digital requerida con la precisión deseada, y almacenar el resultado en un registro de, por decir, ocho bits. A los registros de estatus y datos del convertidor se les asignan direcciones específicas, y se diseña un decodificador para reconocer estas direcciones. Entonces se puede tratar al convertidor A/D como un dispositivo I/O en la forma discutida en la sección 22.2.

■ 23.6 Estándares en la creación de interfases

Existen muchos estándares de *interfacing* que permiten conectar a computadoras, o entre ellos, a dispositivos de diversas rapideces y capacidades de diferentes compañías. En la sección 23.3 se discutió

■ **TABLA 23.2** Resumen de cuatro buses de interfaz estándar.

Atributos ↓	Nombre →	PCI	FireWire	USB
Tipo de bus	Placa madre	I/O paralelo	Serial I/O	Serial I/O
Designación estándar	PCI	ANSI X3.131	IEEE 1394	USB 2.0
Dominio de aplicación típico	Sistema	I/O rápido	I/O rápido	I/O bajo costo
Ancho de bus, bits de datos	36-64	8-32	2	1
Ancho de banda pico, MB/s	133-512	5-40	12.5-50	0.2-15
Número máximo de dispositivos	1 024*	7-31#	63	127 ^s
Alcance máximo, m	<1	3-25	4.5-72 ^s	5-30 ^s
Método de arbitraje	Centralizado	Autoselección	Distribuido	Cadena daisy
Complejidad o costo de transceptor	Alto	Medio	Medio	Bajo

* 32 por segmento de bus.

Uno menos que ancho de bus.

^s Con hubs (repetidores).

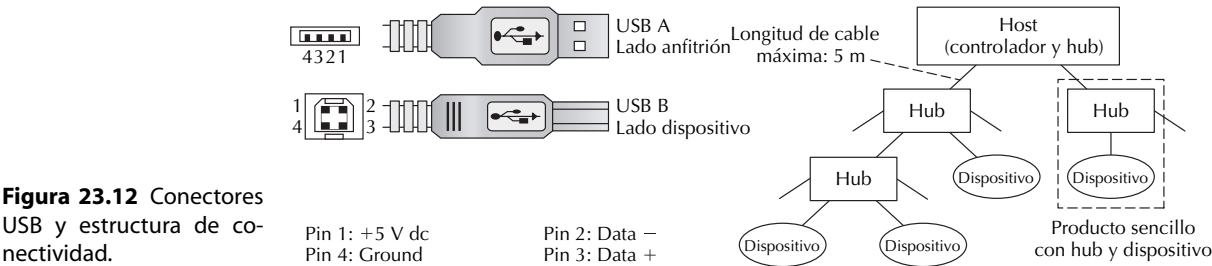


Figura 23.12 Conectores USB y estructura de conectividad.

el bus PCI, y se proporcionó un ejemplo de un protocolo de comunicación de bus. En esta sección se dan algunas otras interfaces de uso común. Para comparación, en la tabla 23.2 se proporciona un resumen de las características de tres de estos estándares, junto con los del PCI.

SCSI (pequeña interfaz de sistema de cómputo; se pronuncia “skuzzzy”) es un bus estándar cuya intención principal son los dispositivos I/O de alta rapidez, como memorias magnéticas y de disco óptico. Creció a partir de la interfaz de sistema Shugart Associates (SASI) a mediados de la década de 1980. Con los años se introdujeron varias versiones de SCSI. La descripción que sigue se basa en SCSI-2, introducido como ANSI Standard X3.131 en 1994, con algunas mejoras y extensiones presentes más tarde. El más reciente SCSI-3 no es estándar sino más bien una familia de estándares. En la mayoría de las PC, el bus IDE/ATA, que se discute al final de la sección, se usa como interfaz para memorias magnéticas y de disco óptico. La razón principal es el bajo costo del último, que supera las mayores capacidades y mayor rendimiento de SCSI.

Los dispositivos en el bus SCSI se comunican con la computadora a través de una de las unidades conectadas al bus. Esta unidad, conocida como controlador, tiene acceso a un bus primario de la computadora, como el PCI y, desde ahí, a memoria. SCSI puede trabajar en modos síncrono y asíncrono, y la tasa de datos es mucho mayor en operación síncrona. SCSI usa conectores de 50 o 68 pines. Las direcciones y datos se multiplexan y el control de bus se efectúa mediante autoselección (sección 23.4). Como consecuencia de lo anterior, el número de dispositivos conectables al bus SCSI es uno menos que su ancho. En el sitio web de SCSI Trade Association, scsita.org, puede encontrar más información acerca de este bus.

USB (*bus serial universal*) es un bus serial estándar pretendido para el uso con dispositivos I/O con ancho de banda de bajo a medio. Los conectores USB usan pequeños enchufes (plug) de cuatro

contactos unidos a un cable de cuatro alambres (figura 23.12). Estos conectores portan tanto señales de datos, en modo medio dúplex, y potencia de 5 V.

Ciertos periféricos de baja potencia pueden recibir energía a través del conector, lo que obvia la necesidad de una fuente de poder separada. Un puerto USB puede alojar hasta 127 dispositivos a través del uso de hubs o repetidores. Se permiten hasta seis tiers, o niveles hub, que forman una cadena en margarita. La computadora anfitriona (*host*) gestiona el bus mediante un protocolo complejo. Cuando un dispositivo se conecta en un bus USB (lo que se puede hacer incluso durante operación), el *host* detecta la presencia del dispositivo y sus propiedades, entonces procede a notificar al sistema operativo la activación del controlador (*driver*) de dispositivo requerido. La desconexión del dispositivo se detecta y maneja de igual manera. Esta capacidad de conexión directa, y la habilidad para soportar muchos dispositivos a través de un solo puerto, forman los atractivos principales de USB. Por ejemplo, una computadora con dos puertos USB puede ligarse a su teclado y ratón mediante un puerto, y a una diversidad de otros dispositivos a través de un hub conectado al segundo puerto (un hub de cinco puertos, por ejemplo, puede alojar una impresora, un *joystick*, una cámara y un escáner).

USB 2.0 tiene tres tasas de datos: 1.5, 12 y 480 Mb/s (0.2, 1.5 y 60 MB/s, respectivamente). El mayor de éstos es específico de “Hi-Speed USB”, un estándar relativamente nuevo. Las otras dos tasas tienen la intención de proporcionar compatibilidad retrospectiva con versiones previas del estándar, incluido USB 1.1, que operaba a 12 Mb/s, e implementaciones de bajo costo para dispositivos que no necesitan altas tasas de datos. En la actualidad, prácticamente toda computadora de escritorio y laptop tiene uno o más puertos USB. En productos más antiguos, se observan puertos USB 1.1; sin embargo, Hi-Speed USB toma ventaja rápidamente. Se puede obtener más información acerca de USB en la literatura [Ande97] o vía el sitio web usb.org.

La interfaz I/O serial IEEE 1394 se basa en la especificación FireWire de Apple Computer, introducida a mediados de la década de 1980. Sony usa el nombre i.Link para esta interfaz. IEEE 1394 representa una interfaz estándar I/O general, aunque su costo relativamente alto en comparación con USB ha limitado su uso a aplicaciones que requieren mayor ancho de banda para tratar con I/O de audio y video de alta calidad. Existen algunas similitudes con USB en la forma y tamaño del conector, longitud máxima de cable y capacidad de conexión directa. Una diferencia clave es el uso de arbitraje de bus par a par que obvia la necesidad de un solo maestro, pero también complica cada interfaz de dispositivo. Soporta transmisiones síncrona y asíncrona. Cualquier dispositivo individual puede solicitar hasta 65% de la capacidad de bus disponible, y hasta 85% de la capacidad se puede asignar a tales dispositivos. El restante 15% de capacidad garantiza que ningún dispositivo será completamente desconectado del uso del bus.

Físicamente, el conector IEEE 1394 tiene seis contactos para potencia más dos pares torcidos. Los últimos están transpuestos en los dos extremos del cable para permitir el uso del mismo conector en ambos extremos (figura 23.13). Cada par torcido está recubierto por separado dentro del cable, y todo el cable también está recubierto en el exterior. En algunos productos también se ha usado un conector más pequeño de cuatro pines, sin líneas de potencia. En el sitio web de la 1394 Trade Association (1394ta.org) se puede obtener más información acerca de IEEE 1394.

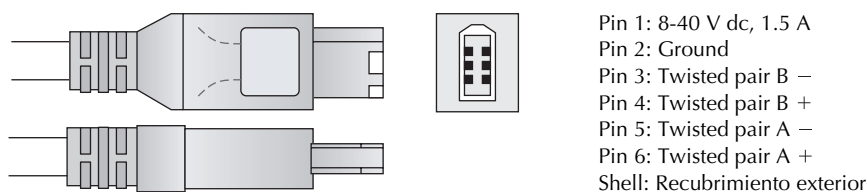


Figura 23.13 Conector IEEE 1394 (FireWire). El mismo conector se usa en ambos extremos.

Otros estándares que uno encuentra con frecuencia son UART, AGP y IDE/ATA. En este contexto, éstos se describen brevemente en los párrafos siguientes.

UART (receptor/transmisor asíncrono universal), oportuno estándar de interfaz serial, es otro nombre que se escucha con frecuencia en conexión con la creación de interfaces. Virtualmente todas las computadoras tienen una unidad UART para gestionar su(s) puerto(s) serial(es). Los caracteres se transmiten en forma similar como en el estándar RS-232, como ya se discutió en la sección 23.1. Sin embargo, la tasa de bits, el ancho de carácter (5-8 bits), el tipo de paridad (ninguna, impar, par) y la secuencia de parada (1, 1.5 o 2 bits) son programables y se pueden seleccionar al momento de configuración. La transición 1 a 0 que inicia el bit de arranque se detecta y forma el punto de referencia para la temporización de los eventos. Si la tasa de transmisión es 19 200 b/s, entonces un nuevo bit se debe muestrear cada $1/19\,200\text{ s} \cong 52.08\text{ }\mu\text{s}$. A una tasa de reloj de muchos megahertz, esto último se puede lograr de manera bastante precisa. Por ejemplo, con un reloj de 20 MHz, la entrada se muestrea cada $20 \times 52.08 = 1\,041$ tics de reloj, y la primera muestra se toma después de 520 tics (mitad del primer tiempo de bit). Alguna imprecisión en la frecuencia de reloj o en la tasa de muestreo son tolerables en tanto el efecto acumulado para el momento en que aparece el bit de parada sea menor que la mitad del tiempo de bit. Por ejemplo, si una muestra se toma cada 1 024 tics en lugar de cada 1 040 tics, el efecto acumulado después de 9.5 tiempos de bit es $9.5 \times 17 = 161.5$ tics = 0.155 tiempo de bit. Lo anterior todavía deja amplio espacio para variaciones en la frecuencia del reloj sin conducir a errores de muestreo. Observe que el bit de arranque tiene una función de sincronización en cuanto a que limita la acumulación de las imprecisiones ya mencionadas más allá del marco temporal de una sola transmisión de carácter. La UART original tenía un buffer de un solo carácter que se llenaría en aproximadamente 1 ms a 9 600 b/s; de modo que, en tanto el CPU respetara una interrupción dentro de 1 ms, no surgiría problema. A las elevadas tasas de transmisión actuales, un buffer de un byte es inadecuado, de modo que UART usualmente se proporciona con un *buffer* FIFO de hasta 128 B para acumular datos e interactuar con el CPU a través de menos interrupciones.

AGP (puerto gráfico acelerado) lo desarrolló Intel para evitar la alta tasa de datos necesaria para representar (*render*) gráficos a partir de sobrecarga del bus PCI. AGP constituye un puerto dedicado en lugar de un bus compartido. Como se muestra en la figura 21.2, tiene una conexión separada hacia el bus de memoria que le proporciona rápido acceso a memoria, donde se pueden mantener ciertos grandes conjuntos de datos asociados con la representación gráfica. AGP usualmente se asigna a un bloque de direcciones contiguas en la parte del espacio de dirección que se encuentra más allá de la memoria física disponible. Una tabla de remapeado de dirección gráfica (GART) traduce las direcciones virtuales AGP en direcciones de memoria física. De esta forma, el controlador de gráficos puede usar una parte de la memoria principal como extensión de su memoria dedicada de video cuando sea necesario. La alternativa al uso de una memoria de video más grande es tanto más costosa como desperdiciada, porque la memoria dedicada puede pasar sin usarse en algunas aplicaciones.

Los conectores AGP tienen 124 o 132 pines, para voltaje sencillo (1.5 o 3.3 V) y variedades universales, respectivamente. El ancho de banda pico varía de 0.25 a 1 GB/s. El elevado ancho de banda se logra, en parte, con ayuda de ocho bits de dirección “laterales” adicionales que permiten a AGP seguir nuevas direcciones y solicitudes mientras los datos se transfieren en las líneas principales de dirección/datos de 32 bits. Desde luego, el elevado ancho de banda AGP es útil sólo si el bus de sistema y la memoria principales pueden soportar un ancho de banda un poco mayor, para permitir que tanto AGP como CPU accedan a memoria.

ATA, o adjunto AT, se llama así porque se desarrolló como un adjunto a la primera computadora personal IBM, PC-AT. ATA, y sus descendientes como Fast ATA y Ultra ATA, se usan comúnmente para hacer interfaz con unidades de almacenamiento para PC. IDE (electrónica de dispositivo integrado) es una marca registrada para los productos ATA de Western Digital; por tanto, es virtualmente

equivalente a ATA. Este último usa un conector de 40 pines, que usualmente se une a un cable plano flexible de 40 alambres con dos conectores. Como ya se mencionó, esta interfaz más bien limitada ha sido superada en ancho de banda y otras capacidades por otros estándares. Sin embargo, ATA sobrevive y todavía está en uso primario debido a su simplicidad y bajo costo de implementación.

PROBLEMAS

23.1 Interfaz serial RS-232

Responda las siguientes preguntas acerca de la interfaz serial RS-232 con base en investigación en libros y en Internet.

- ¿Cómo se usan en el protocolo de *handshaking* las diversas señales de control que se muestran en la figura 23.3?
- ¿Cómo se direcciona o selecciona un dispositivo en la figura 23.2a?
- ¿Qué tipos de dispositivos de entrada y salida puede soportar esta interfaz?
- ¿Cómo se establece conexión virtual a través del intercambio de mensajes y reconocimientos?
- ¿Qué categorías de calidad de servicio se soportan?
- ¿Cuántos canales de audio y video se pueden soportar dentro de las tasas de datos ATM mencionadas cerca del final de la sección 23.1?
- ¿Bajo qué condiciones caen las celdas ATM?

23.2 Colisiones en Ethernet

Considere el siguiente análisis muy simplificado de colisiones en Ethernet. Un cable Ethernet interconecta n computadoras. El tiempo se divide en rebanadas de 1 ms. En un intervalo de 0.5 s, cada una de las n computadoras intenta transmitir un mensaje, y la temporización de la transmisión se selecciona aleatoriamente de entre las 500 rebanadas de tiempo disponibles. Sin importar si la transmisión es exitosa, el transmisor se rinde y no hace otro intento de transmisión en el actual intervalo de 0.5 s.

- ¿Cuál es la probabilidad de tener uno o más conflictos en un intervalo de 0.5 s con $n = 60$?
- ¿Cómo se relaciona el resultado de la parte a) con el “problema de cumpleaños” descrito en la sección 17.4, en conexión con memorias interpoladas?
- En promedio, dentro de cada intervalo de 0.5 s, ¿qué fracción de las transmisiones intentadas de la parte será exitosa?
- Repita la parte a), esta vez suponiendo $n = 150$.
- Repita la parte c) con la suposición de la parte d).

23.3 Modo de transferencia asíncrono

Responda las siguientes preguntas acerca de la tecnología de interconexión ATM con base en investigación en libros y en Internet.

23.4 Protocolos de bus

En un bus, las líneas de dirección y datos se pueden separar o compartir (multiplexar). ¿Cuál de estas dos opciones es probable que se suponga en los siguientes buses y por qué?

- El bus síncrono cuya operación se muestra en la figura 23.6
- El bus asíncrono cuya operación se muestra en la figura 23.7
- El bus del ejemplo 23.1

23.5 Buses síncronos

Considere el bus síncrono cuya operación de muestra en la figura 23.6. Suponga que todas las transacciones de bus son del mismo tipo e ignore la cabecera de arbitraje.

- ¿Qué fracción del ancho de banda pico del bus se usa realmente para transferencias de datos?
- ¿Cómo se mejora el ancho de banda disponible si se usa un protocolo de transacción dividida?

23.6 Puentes Ethernet

Las redes Ethernet grandes por lo general se forman con segmentos más pequeños conectados mediante puentes. Se puede usar un *puente transparente* (también conocido como *puente de árbol de expansión*) para expandir una red sin modificación alguna al hardware o software de los

usuarios existentes. Estudie cómo operan estos puentes transparentes y cómo usan un algoritmo de aprendizaje retrospectivo para construir sus tablas de enrutamiento, que están en blanco al momento de arranque. Describa sus hallazgos en un informe de cinco páginas.

23.7 Operación de escritura I/O en bus PCI

- Dibuje un diagrama similar al de la figura 23.8 para mostrar una operación de escritura I/O vía bus PCI.
- Explique los elementos de su diagrama en forma similar a la de la operación de lectura I/O que aparece al final de la sección 23.3.

23.8 Árbitros de prioridad fija

Se puede diseñar un árbitro de prioridad fija en forma muy parecida a un sumador con acarreo en cascada, con propagación de señal de “permiso” a partir de la prioridad más alta hacia el lado de la prioridad más baja y detenerse en la primera solicitud, si hay alguna.

- Presente el diseño completo de un árbitro de prioridad de tipo cascada con cuatro señales de solicitud y cuatro señales de garantía.
- Muestre cómo se puede hacer más rápido el árbitro mediante técnicas de adelantamiento.
- ¿Cómo se puede construir un árbitro de prioridad de ocho entradas con el uso de dos árbitros del tipo definido en las partes a) o b)?

23.9 Esquemas de prioridad sin inanición

Un problema con un esquema de arbitraje de prioridad fija es que los dispositivos de prioridad baja nunca pueden tener oportunidad de usar el bus si las unidades de prioridad más alta colocan una fuerte demanda sobre él. A esta condición se le conoce como inanición.

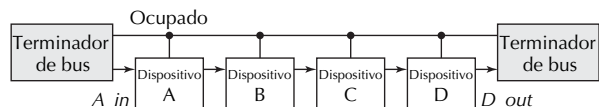
- Diseñe un árbitro de prioridad rotatoria con cuatro señales de solicitud y cuatro señales de garantía. El dispositivo de mayor prioridad cambia cada vez que el bus se garantiza a un dispositivo, de modo que cada dispositivo tiene una oportunidad de convertirse en el dispositivo de mayor prioridad, ello garantiza no inanición.
- El esquema de prioridad rotatorio de la parte a) no es un esquema de prioridad genuino en cuanto a que todos los dispositivos son tratados igualmente. Im-

plemente un árbitro de prioridad de cuatro entradas en el que los dispositivos de mayor prioridad tengan más posibilidad de usar el bus, aunque los dispositivos de baja prioridad no tengan inanición. *Sugerencia:* ¿Y si se permite a cada dispositivo al menos una oportunidad de usar el bus después de que el siguiente dispositivo de mayor prioridad ha garantizado el uso del bus x veces, donde x es un parámetro ajustable del diseño?

- Sugiera e implemente un árbitro diferente con la propiedad sugerida en la parte b), pero ahora el criterio para la decisión de garantía es el número de veces que se niega la solicitud de bus de un dispositivo

23.10 Arbitraje de bus distribuido

Considere el siguiente esquema de arbitraje de bus distribuido. Cada dispositivo X tiene una señal de control de entrada X_{in} y una señal de control de salida X_{out} y puede postular y observar la señal *busy* (ocupado) del bus. La entrada de control al dispositivo de la extrema izquierda siempre se postula, ello le da la mayor prioridad. Al comienzo de cada ciclo de reloj, un dispositivo puede despostular su señal de salida de control, que sirve como indicador de solicitud de bus. Si X_{in} se postula, el dispositivo X debe postular X_{out} si no pretende usar el bus. Cuando la propagación de señal se completa, el único dispositivo con su entrada de control postulada y su salida de control despostulada tiene prioridad para usar el bus al comienzo del siguiente ciclo de bus. Discuta consideraciones de diseño para este esquema de prioridad distribuida, incluida la relación entre el ciclo de reloj y el tiempo de propagación de señal en la cadena de dispositivos.



23.11 Arbitraje de bus distribuido

La serie VAX de computadoras usaba un bus con arbitraje distribuido y 16 líneas de solicitud de bus R_i , $0 \leq i \leq 15$, donde las líneas de índice más bajo tenían mayor prioridad. Cuando el dispositivo i quería usar el bus, hacía una reservación para una rendija futura al postular R_i durante la rendija de tiempo actual. Al final de la ren-

dija de tiempo actual, todos los dispositivos solicitantes examinaban las líneas de solicitud para determinar si un dispositivo de mayor prioridad tenía solicitado el bus; si no, entonces el dispositivo podía usar el bus durante la rendija solicitada.

- Demuestre cómo los 17 dispositivos pueden compartir el bus aun cuando sólo haya 16 líneas de solicitud.
Sugerencia: El dispositivo adicional tendría la prioridad más baja.
- Argumente que, usual y paradójicamente, el dispositivo de prioridad más baja de la parte *a*) tiene el tiempo de espera promedio más corto para usar el bus. Por ende, en VAX, esta mancha de prioridad más baja se asignó al CPU.
- Discuta condiciones bajo las cuales la postulación de la parte *b*) no se sostendría.

23.12 Rendimiento de bus

- Vuelva a hacer el ejemplo 23.3, pero esta vez suponga que las transacciones de bus toman 5 ns (10% de los casos), 25 ns (20%), 35 ns (25%) y 55 ns (45%). Proporcione justificación intuitiva para cualquier diferencia entre sus resultados y los obtenidos en el ejemplo 23.3.
- Repita la parte *a*) con los siguientes tiempos y porcentajes de transacciones: 5 ns (30%), 30 ns (50%), 55 ns (20%).

23.13 Buses para detección e instrumentación

Los buses que se usan para comunicar datos entre sensores, instrumentos y una computadora de control se conocen como *fieldbuses* (buses de campo o experimentales). Un estándar de *fieldbus* usado comúnmente es el fieldbus Foundation. Responda las siguientes preguntas acerca del estándar de fieldbus Foundation con base en investigación en libros y en Internet.

- ¿Cuáles son los criterios principales de diseño que hacen diferente a un *fieldbus* de otros buses?
- ¿Cómo se conectan los dispositivos al bus y cuál es el método de arbitraje, para el caso de que haya alguno?
- ¿Cuál es el ancho de datos del bus y cómo se transmiten los datos sobre éste?
- ¿Cuáles son los dominios de aplicación más importantes para el bus?

23.14 Conversión A/D y D/A

- Una forma de convertir una señal digital de cuatro bits a un nivel de voltaje análogo en el rango de, por decir, 0-3 V, consiste en conectar una red de resistores en escalera a las cuatro salidas de un registro que contiene la muestra digital. Presente un diagrama de tal convertidor D/A y explique por qué produce la salida deseada.
- Conceptualmente, la forma más simple de convertir un voltaje de entrada análogo en el rango de, por decir, 0-3 V, es usar los llamados circuitos flash, que consisten de resistores, comparadores de voltaje y un codificador de prioridad. Estudie el diseño de los convertidores flash A/D y prepare un informe de dos páginas acerca de cómo funcionan.
- Cuando la entrada al convertidor A/D de la parte *b*) es un voltaje que varía rápidamente, el muestreo también debe capturar la frecuencia de la señal además de su amplitud. Discuta brevemente cómo se determina un muestreo adecuado de frecuencia. Para esta parte, consulte un texto o manual de procesamiento de señales digitales.

23.15 Fundamentos de creación de interfases

Suponga que el convertidor A/D usado en la aplicación de la veleta de la figura 23.11 es más bien imprecisa pero no tiene un error o sesgo sistemático. En otras palabras, el error es verdaderamente aleatorio y puede ser en cualquier dirección. Subraye un procedimiento que permite al microcontrolador compensar el error de conversión para que permita que se obtengan resultados más precisos para la dirección del viento.

23.16 Otros estándares de creación de interfases

Muchos estándares de *creación de interfases* más antiguos, menos conocidos o específicos de dominio, así como más detallados de los estándares más ampliamente usados, no se discutieron en la sección 23.6. Algunos de éstos se mencionan a continuación. Prepare un informe de dos páginas acerca de las características y dominios de aplicaciones de uno de los siguientes interfases estándar.

- Bus ISA (arquitectura estándar de industria).
- Bus VESA VL-local (extensión de ISA).
- Tecnología plug and play para PCI.

- d) Puerto paralelo.
- e) PCMCIA para computadoras notebook.

23.17 Bahía universal en computadoras notebook

Algunas computadoras notebook tienen una bahía universal (a veces llamada multibahía) en la que se pue-

den insertar una variedad de dispositivos I/O, o incluso una segunda batería. Especule acerca de cómo se puede conectar a la computadora (pines, conectores, etc.) el dispositivo insertado, cómo la computadora puede reconocer qué tipo de dispositivo está presente en la bahía y por qué tal cambio de dispositivo no afecta la programación I/O.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Alex93] Alexandridis, N., *Design of Microprocessor-Based Systems*, Prentice Hall, 1993.
- [Ande97] Anderson, D., *Universal Serial Bus System Architecture*, Addison-Wesley (Mindshare), 1997.
- [Bate02] Bateman, A. y I. Paterson-Stephens, *The DSP Handbook: Algorithms, Applications, and Design Techniques*, Prentice Hall, 2002.
- [Buch00] Buchanan, W., *Computer Busses*, Arnold, 2000.
- [Dudr96] Dudra, F., “Serial and UART Tutorial”. Vea http://freebsd.unixtech.be/doc/en_US.ISO8859-1/articles/serial-uart/
- [Lobu02] LoBue, M. T., “Surveying Today’s Most Popular Storage Interfaces”, *IEEE Computer*, vol. 35, núm. 12, pp. 48-55, diciembre de 2002.
- [Myer02] Myers, R. L., *Display Interfaces: Fundamentals and Standards*, Wiley, 2002.
- [Ston82] Stone, H. S., *Microcomputer Interfacing*, Addison-Wesley, 1982.
- [Wolf01] Wolf, W., *Computers as Components: Principles of Embedded Computing System Design*, Morgan Kaufmann, 2001.
- [WWW] Las siguientes organizaciones presentan información actualizada acerca de varios buses y estándares vistos en este capítulo: ansi.org, atmforum.com, pcisig.com, scsita.org, standards.ieee.org, usb.org.

■ CAPÍTULO 24

CONMUTACIÓN CONTEXTUAL E INTERRUPCIONES

“Cuando tu trabajo habla por sí mismo, no interrumpas.”

Henry J. Kaiser

“Como decía el otro día...”

Luis Ponce de León, al resumir una conferencia interrumpida por cinco años de prisión

TEMAS DEL CAPÍTULO

- 22.1** Peticiones al sistema por I/O
- 22.2** Interrupciones, excepciones y trampas
- 22.3** Manejo de interrupciones simples
- 22.4** Interrupciones anidadas
- 22.5** Tipos de conmutación contextual
- 22.6** Hilos y multihilos

Los usuarios y programadores comunes usualmente no necesitan preocuparse por los detalles de la operación de los dispositivos I/O y cómo controlarlos. Estas tareas se relegan al sistema operativo, al que los programas usuarios llaman siempre que se necesite I/O. A su vez, el sistema operativo activa los controladores del dispositivo que generan procesos I/O asíncronos que realizan sus tareas y reportan de vuelta al CPU mediante interrupciones. Cuando se detecta una interrupción, el CPU puede cambiar el contexto, hacer a un lado el programa activo y ejecutar una rutina de atención de interrupción. Este tipo de conmutación contextual también es viable entre diferentes programas usuarios, o entre hilos del mismo programa, como un mecanismo para evitar largas esperas debidas a la indisponibilidad de datos o recursos de sistema.

■ 24.1 Peticiones al sistema por I/O

Como se mencionó en la sección 7.6, y de nuevo al final de la sección 22.2, la mayoría de los usuarios no se involucran en detalle de transferencias I/O entre dispositivos y memoria, sino que usan peticiones de sistema para lograr transferencias de datos I/O. Una razón para I/O indirecta a través del sistema operativo constituye la necesidad de proteger datos y programas contra daño accidental o deliberado. Otra significa que es más conveniente para el usuario, así como para el sistema, apoyarse en rutinas de sistema operativo para proporcionar una interfaz clara y uniforme a tales dispositivos en una forma que sea, en la

medida de lo posible, independiente de dispositivo. Una rutina de sistema inicia la I/O, que usualmente se realiza mediante DMA, después cede el control al CPU. Más tarde, este último recibirá una interrupción que significa la conclusión de I/O y permitirá que la rutina I/O realice su etapa de limpieza.

Por ejemplo, al leer datos de un disco, el usuario debe evitar acceder o modificar datos o archivos del sistema que pertenezcan a otro usuario. Adicionalmente, durante una operación de lectura se pueden encontrar una docena o más tipos de errores. Usualmente cada uno de los errores tiene un *bit flag* asociado en el registro de estatus del dispositivo. Los ejemplos incluyen número inválido de cilindro (pista, sector), violación de *checksum*, temporización anómala y dirección de memoria inválida. Estos bits de estatus se deben verificar en cada operación I/O e iniciar acción apropiada para lidiar con cualquier anomalía. Tiene perfecto sentido liberar al usuario de tales complejidades al proporcionar rutinas de atención I/O, cuyas interfases de usuario se enfoquen en la transferencia de datos requeridas y no en la mecánica de la transferencia en sí misma.

La conveniencia e independencia de dispositivo se logran al proporcionar varias capas de control entre el programa usuario y los dispositivos I/O de hardware. Las capas típicas incluyen:

Kernel OS → Subsistema I/O → *Driver* de dispositivo → Controlador de dispositivo → Dispositivo

El *driver* del dispositivo es una liga de software entre los dos componentes hardware a la derecha y el sistema operativo y su subsistema I/O a la izquierda. Como tal, es dependiente tanto de OS como de dispositivo. Un nuevo dispositivo I/O que no siga una interfaz hardware/software ya establecida se debe introducir en el mercado con *drivers* de dispositivo para varios sistemas operativos populares. La idea es capturar, tanto como sea posible, las características de un dispositivo I/O en su *driver* asociado, de modo que el subsistema I/O necesite lidiar sólo con ciertas categorías generales, y que cada una abarque muchos dispositivos I/O.

El sistema operativo (OS, por sus siglas en inglés) representa el software de sistema que controla todo en una computadora. Sus funciones incluyen gestión de recursos, cronograma de tareas, protección y manejo de excepción. El subsistema I/O dentro del sistema operativo maneja las interacciones con dispositivos I/O y gestiona las transferencias de datos reales. Con frecuencia consta de una capa básica que está estrechamente involucrada con características de hardware y otros módulos de apoyo que pueden no requerirse para todos los sistemas. El subsistema I/O básico para el caso del sistema operativo Windows es el BIOS (*basic input/output system*) que, además del I/O básico, tiene cuidado de autocargar (*booting*) el sistema en el encendido inicial. Las rutinas BIOS, por lo general, se almacenan en ROM o memoria flash para evitar pérdidas en el evento de interrupción de energía.

Al agrupamiento antes mencionado de dispositivos I/O en un pequeño número de tipos genéricos se le conoce como abstracción I/O. En lo que sigue, se revisan cuatro de las abstracciones I/O más útiles que se encuentran usualmente en los sistemas operativos modernos:

Flujo de caracteres I/O.

Bloque de I/O.

Sockets de red.

Relojos o temporizadores.

La abstracción de flujo de caracteres I/O trata la entrada o salida como una corriente de caracteres. Un nuevo carácter de entrada se agrega al final de la corriente de entrada, mientras que otro nuevo carácter de salida se adelanta al dispositivo de salida después del carácter previo. Este tipo de comportamiento es capturado por las peticiones de función de sistema `get (·)` y `put (·)`, donde “get” obtiene un carácter de entrada y “put” envía un carácter a la salida. Con el uso de estos primitivos, es relativamente sencillo construir rutinas I/O que ingresen o saquen tiras de muchos caracteres. De hecho, al examinar las primeras ocho líneas de la tabla 7.2 se observa que el I/O orientado a carácter es el único tipo proporcionado en

MiniMIPS (y su simulador) al nivel del lenguaje ensamblador. El flujo de caracteres I/O es particularmente adecuado para dispositivos como teclados, ratones, módems, impresoras y tarjetas de audio.

La abstracción bloque de I/O es adecuada para discos y otros dispositivos orientados a bloque. En este sentido, tres peticiones básicas de sistema pueden capturar la esencia del bloque de I/O: `seek(•)`, `read(•)` y `write(•)`. La primera de éstas, *seek*, se necesita para especificar cuál bloque se debe transferir, mientras que *read* y *write* realizan la transferencia de datos I/O real hacia y desde memoria, respectivamente. Es posible construir un sistema de archivos mapeados a memoria en lo alto de tales primitivos I/O orientados por bloque. La idea es que la petición de sistema para I/O regrese la dirección virtual de los datos requeridos en lugar de iniciar la transferencia de datos. En consecuencia, los datos se cargan automáticamente en memoria con el primer acceso a ellos a través de la dirección virtual proporcionada. De esta forma, el acceso al sistema de archivos se vuelve muy eficiente porque sus transferencias de datos se manejan mediante el sistema de memoria virtual.

Los sockets de red son dispositivos I/O especiales que soportan comunicación de datos a través de redes de computadoras. Una petición de sistema I/O a este respecto puede crear un socket, hacer que un socket local se conecte a una dirección remota (un socket creado por otra aplicación), detectar aplicaciones remotas que solicitan ser conectados en un socket local, o enviar/recibir paquetes. Los servidores, que usualmente soportan muchos sockets, adicionalmente pueden requerir instalaciones para detección más eficiente de cuáles sockets tienen paquetes en espera de ser recibidos y cuáles tienen espacio para aceptar un nuevo paquete para transmisión.

Los relojes y temporizadores, que son componentes esenciales en la implementación de aplicaciones de control de tiempo real con microcontroladores, también se encuentran en procesadores de propósito general en vista de su utilidad para determinar el tiempo actual del día, medir tiempo transcurrido e impulsar eventos en tiempos preestablecidos. Un *temporizador de intervalo programable* constituye un mecanismo de hardware que se puede preestablecer a un espacio de tiempo deseado y generar una interrupción cuando dicha cantidad de tiempo haya transcurrido. El sistema operativo usa temporizadores de intervalo para programar con base en un cronograma tareas periódicas o asignar una *rebanada de tiempo* a una tarea en un ambiente multitarea. También puede permitir a los procesos de usuario utilizar esta facilidad. En el último caso, la solicitud del usuario para temporizadores de intervalo más allá del número disponible en hardware puede ser atendido al establecer temporizadores virtuales. La cabecera de mantener estos temporizadores virtuales es pequeña en comparación con los intervalos que se miden en aplicaciones típicas. Por ejemplo, un espacio de tiempo de 1 ms corresponde a 10^6 tics de reloj con un reloj de 1 GHz.

■ 24.2 Interrupciones, excepciones y trampas

Ya se discutió (secciones 22.4 y 22.5) cómo se usan las interrupciones para liberar al CPU de la microgestión del proceso de transferencia de datos I/O. En ésta y en las dos secciones siguientes, se tratará acerca de cómo se implementan y manipulan las interrupciones.

Las interrupciones en una computadora se parecen mucho a las interrupciones de teléfono y correo electrónico durante el tiempo de trabajo o de estudio de una persona (figura 24.1). Cuando un teléfono suena, se hace a un lado lo que se está haciendo (marcar una página que se lee en un libro, golpear el botón guardar en la computadora, presionar el botón *mute* en la TV) y platicar con quien llama. Después de colgar se pueden escribir algunas notas acerca de lo que se necesita hacer como resultado de la llamada telefónica y regresar al trabajo principal. Los estudios demuestran que toma entre uno y dos minutos “recuperarse” de la interrupción y continuar trabajando a la misma tasa anterior a que el teléfono sonara. Algo similar se aplica cuando se tiene una alerta de correo electrónico, aunque, por razones desconocidas, acaso tengan que ver con la relativa dificultad de leer y escribir frente a hablar,

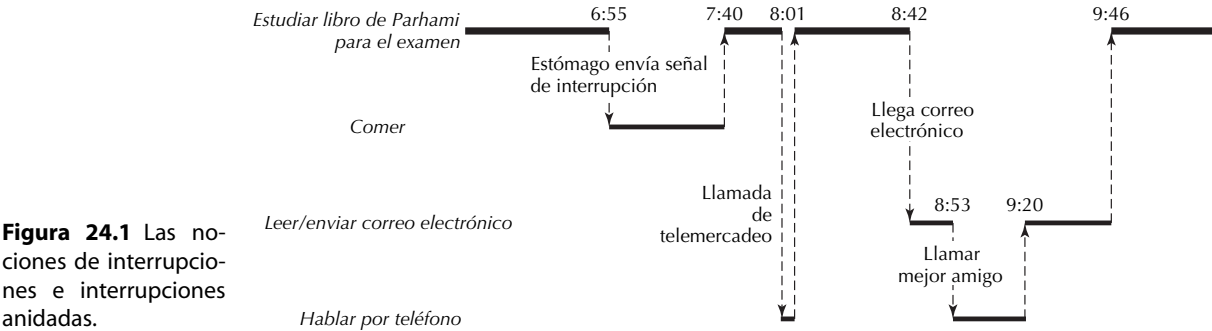


Figura 24.1 Las nociones de interrupciones e interrupciones anidadas.

el tiempo de recuperación es poco mayor en este caso. Se pueden evitar o deshabilitar completamente tales interrupciones si se desconecta el cordón del teléfono o se revisa el correo electrónico cada dos horas, en lugar de continuamente. Esta situación es similar a lo que hace el CPU cuando deshabilita las interrupciones o las restringe sólo a las de alta prioridad.

La *interrupción* es tanto el término general usado para un CPU que desvía su atención de la tarea actual que se ejecuta hacia algún evento inusual o impredecible que demanda su involucramiento (por cualquier razón) y el tipo específico de interrupción causado por entrada/salida y otras unidades de hardware. En este contexto, a veces se habla de *interrupciones de hardware*. El CPU se puede desviar de la tarea actual por causas en otras dos categorías:

Excepciones, que son causadas por operaciones ilegales, como dividir entre 0, intentar ingresar a localidad de memoria no existente o ejecutar instrucciones privilegiadas en modo usuario. Las excepciones son impredecibles por naturaleza y más bien infrecuentes.

Trampas, o interrupciones de software, que son peticiones deliberadas a sistema operativo cuando se necesitan servicios particulares. A diferencia de las interrupciones de hardware y las excepciones, las trampas están preplaneadas en el diseño de un programa y no son raras.

Gran parte de nuestra discusión de este capítulo trata de la manipulación de caminos de interrupción de hardware. Sin embargo, las excepciones y trampas son manipuladas de manera semejante en estas requeridas sin un lado de inicio de ejecución del programa y llamada a una rutina de software especial para tratar con la causa.

En el esquema más simple existe sólo una petición de la línea interrupción dentro del CPU. Múltiples dispositivos que requieren interrumpir el CPU comparten de esta línea. Manipular las interrupciones en esta mínima configuración se discute en la sección 24.3. De forma más general, existen múltiples líneas de solicitud de interrupción de varias prioridades. El siguiente ejemplo muestra por

Ejemplo 24.1: Niveles de prioridad de interrupciones en la computadora de un automóvil Suponga que en un automóvil se usa un microcontrolador que tiene cuatro niveles de prioridad para interrupciones. Las unidades que interrumpen al controlador incluyen un subsistema de detección de impacto, que necesita atención dentro de 0.1 ms, junto con otros cuatro subsistemas, a saber:

Subsistema	Tasa de interrupción	Máx tiempo atención (ms)
Combustible/encendido	500/s	1
Temperatura de motor	1/s	100
Pantalla de tablero	800/s	0.2
Acondicionador de aire	1/s	100

Discuta cómo se pueden asignar las prioridades para garantizar el tiempo de respuesta de 0.1 ms para el detector de impacto y manejar cada una de las otras unidades críticas antes de que se produzca la siguiente interrupción.

Solución: Al detector de impacto se le debe dar prioridad pues su tiempo de respuesta requerido es menor que el tiempo de atención para cada uno de los otros tipos de interrupción; ningún otro subsistema puede compartir este alto nivel de prioridad. Al acondicionador de aire se le puede dar menor prioridad porque su función no es crucial para la seguridad. Todavía quedan dos niveles de prioridad y tres subsistemas. Como consecuencia de que el monitor de temperatura del motor tiene un tiempo de atención máximo de 100 ms, su prioridad es baja, similar a la de los subsistemas de pantalla y combustible/encendido, que necesitan muchos cientos de periodos de servicios por segundo. Finalmente, los últimos pueden compartir el mismo nivel de prioridad, pues que cada uno se puede atender después del otro sin exceder el tiempo disponible antes de la siguiente interrupción. Por ejemplo, si la atención del subsistema combustible/encendido comienza justo antes de la interrupción de pantalla, transcurrirán 1.2 ms antes de que ambas rutinas de atención de interrupción se completen, mientras que hay $1/800 \text{ s} = 1.25 \text{ ms}$ disponibles antes de que llegue la siguiente interrupción de pantalla de tablero. Observe que se ignoraron el tiempo de atención para las interrupciones del subsistema de detección de impacto que tiene prioridad, porque ocurren muy rara vez; cuando lo hacen, lo demás carece de importancia.

qué puede ser útil tener múltiples niveles de prioridad para interrupciones. Las interrupciones anidadas y los conflictos en su manejo se discuten en la sección 24.4.

24.3 Manejo de interrupciones simples

En esta sección se supone una sola línea de interrupción que va hacia, y una señal de reconocimiento de interrupción producida por, el CPU. En la sección 24.4 se discutirán múltiples líneas de interrupción con niveles de prioridad jerárquicos y el anidado asociado de actividades de atención de interrupción.

Las señales de solicitud de interrupción de todas las unidades participantes están conectadas a la línea IntReq en el CPU (figura 24.2). Hay un flip-flop *interrupt mask* (máscara de interrupción) que el CPU puede fijar para deshabilitar interrupciones. Si IntReq se postula y las interrupciones no se enmascaran, se fija un flip-flop dentro del CPU para registrar la presencia de una solicitud de interrupción y para:

Reconocer la interrupción mediante la postulación de la señal IntAck.

Notificar a la lógica de siguiente dirección del CPU que está pendiente una interrupción.

Establecer la máscara de interrupción de modo que no se acepte ninguna nueva interrupción.

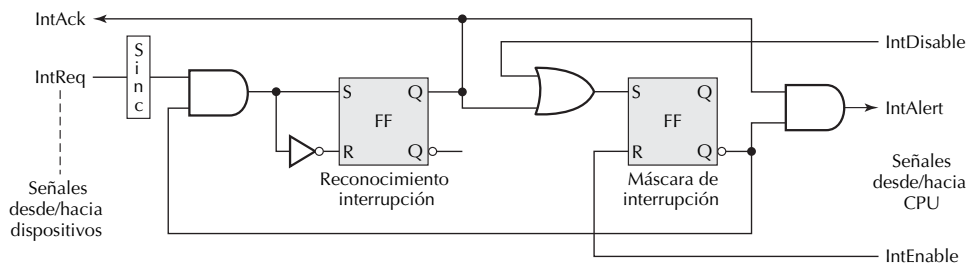


Figura 24.2 Lógica de interrupción simple para el MicroMIPS de ciclo sencillo.

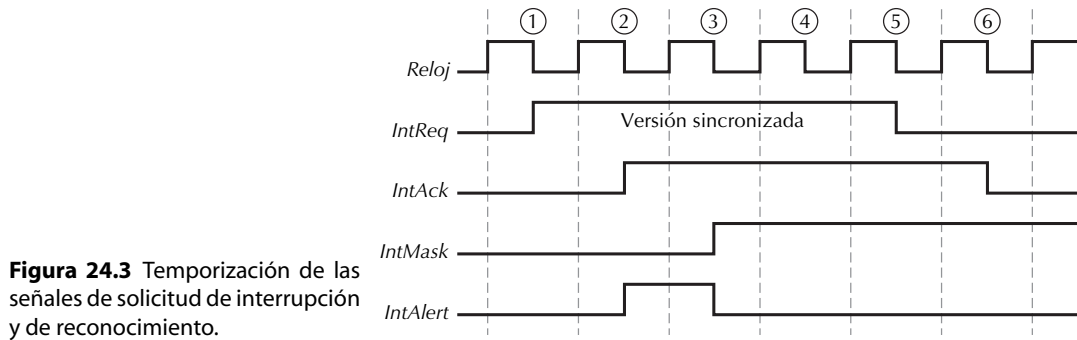


Figura 24.3 Temporización de las señales de solicitud de interrupción y de reconocimiento.

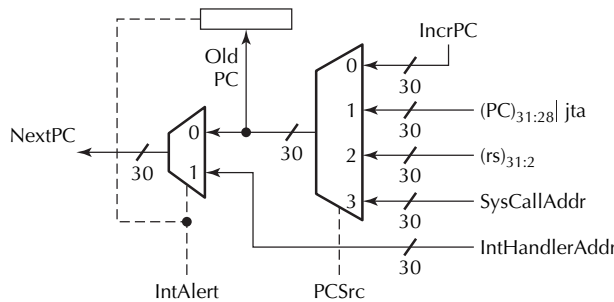


Figura 24.4 Parte de la lógica de siguiente dirección para MicroMIPS de ciclo sencillo, con capacidad de interrupción añadida (comparar la parte inferior izquierda de la figura 13.4).

Luego que los dispositivos notan la postulación de la señal *IntAck*, despostulan sus señales de solicitud, ello conduce al restablecimiento del flip-flop de reconocimiento de interrupción. Esta secuencia de eventos se muestra en la figura 24.3. Mientras tanto, después del reconocimiento de interrupción y antes de que se fije la máscara de interrupción, se produce una señal de alerta de interrupción (*IntAlert*) que ordena al CPU iniciar la ejecución de una rutina de interrupción. Lo anterior se hace al cargar la dirección de inicio del manipulador de interrupción en el PC y salvar el contenido previo del PC a usarse como la dirección de retorno cuando se atendió la interrupción.

Considere, por ejemplo, cómo se puede añadir tal capacidad de interrupción simple a la implementación MicroMIPS de ciclo sencillo del capítulo 13. La figura 24.4 muestra la parte modificada en la lógica de siguiente dirección de la figura 13.4 para permitir que la dirección de una rutina de manipulación de interrupción se cargue en PC siempre que se postule la señal *IntAlert*. Desde luego, el contenido de PC se debe salvar y usar como la dirección de retorno después de que la interrupción se atienda. Se puede suponer la inclusión de un registro especial que se cargue con el contenido antiguo (original) del PC siempre que se postule *IntAlert*. Entonces es necesario proporcionar una instrucción “retorno de interrupción” que coloque el contenido PC salvado en el PC y postule la señal *IntEnable* de modo que las nuevas interrupciones se acepten nuevo. Observe que no se puede usar el registro *\$ra* para salvar la dirección de retorno de interrupción como se hizo para los procedimientos. La razón es que una interrupción puede ocurrir en cualquier momento, incluso dentro de un procedimiento que todavía no haya salvado (o no necesite salvar) su dirección de retorno en la pila.

Puesto que sólo hay una señal de interrupción en el CPU, la primera orden del negocio para el manipulador de interrupciones consiste en determinar la causa de la interrupción de modo que se puedan tomar acciones adecuadas. Una forma de hacer esto último es que el manipulador de interrupciones sondee todos los dispositivos I/O para ver cuál solicitó atención. El procedimiento para ello es similar al sondeo de I/O discutido en la sección 22.3. El sondeo puede ocurrir, y la atención se puede brindar, en el orden de prioridades de dispositivo. Una alternativa al sondeo es proporcionar la señal de recono-

cimiento de interrupción sólo al dispositivo de mayor prioridad y usar una cadena en margarita para pasar la señal a otros dispositivos en orden descendente de prioridades. Esto es similar al adelantamiento de la señal de garantía de bus en el ordenamiento de la figura 23.10. Un dispositivo que solicite atención puede enviar su identidad sobre el bus de dirección o de datos al recibir la señal de reconocimiento del CPU. Cualquier dispositivo de prioridad baja no verá la señal de reconocimiento de interrupción y continuará postulando su señal de solicitud de interrupción hasta que reciba un reconocimiento. Observe que el sondeo es más flexible que el encadenamiento en margarita en cuanto que permite que las prioridades se modifiquen fácilmente. La desventaja del sondeo es que desperdicia mucho tiempo interrogando muchos dispositivos I/O siempre que alguno de ellos genera una interrupción.

Luego que el CPU aprende la identidad del dispositivo (de mayor prioridad) que solicitó la interrupción, salta hacia el segmento apropiado de la rutina de atención de interrupción que maneja el tipo de solicitud particular. Este proceso se puede hacer más eficiente al permitir al dispositivo suministrar la dirección de inicio para su manipulador de interrupción deseado; de esta forma, la transición hacia dicho manipulador puede ocurrir inmediatamente. Un beneficio de este método, que se conoce como *interrupción vectorizada*, consiste en que un dispositivo puede proporcionar diferentes direcciones dependiendo de su tipo de solicitud; por tanto, evita interrogaciones o niveles adicionales de rodeos. En cualquier caso, la rutina de atención de interrupción debe salvar y, al terminar, restaurar cualquier registro que necesite usar en el curso de su ejecución. Esto es muy parecido a lo que ocurre en un procedimiento, excepto que aquí todos los registros tienen el mismo estatus en relación a salvar y restaurar (a un procedimiento ordinario se le permite usar algunos registros sin salvarlos; vea las convenciones de uso de registro MiniMIPS en la figura 5.2).

El ordenamiento anterior se puede extender a múltiples niveles de interrupciones con diferentes prioridades mediante el uso de un codificador de prioridad (figura 24.5). Cada línea de solicitud de interrupción a la izquierda representa un conjunto de dispositivos con prioridades idénticas. Si se reciben una o más solicitudes de interrupción, se postula la señal *IntReq* para notificar al CPU. Adicionalmente, al CPU se proporciona la identidad de la línea de solicitud de mayor prioridad postulada. De esta forma, o el CPU puede identificar únicamente al dispositivo interruptor (cuando una línea de solicitud representa un solo dispositivo) o de otro modo sondeará un subconjunto de todos los dispositivos. Esto es más eficiente que tener que sondear todos los dispositivos con cada interrupción. Si el CPU debe habilitar o deshabilitar cada tipo de interrupción por separado, se debe modificar el esquema que se muestra en la figura 24.5 para mover la capacidad de máscara de interrupción al lado de entrada del codificador de prioridad. Se deja a los lectores proporcionar los detalles de la modificación requerida.

La implementación MicroMIPS de ciclo sencillo considerada hasta el momento da seguimiento continuamente a la condición de interrupción y comienza a ejecutar la rutina de atención de interrupción en el ciclo de reloj siguiente a la postulación de *IntAlert*. Agregar capacidad de interrupción a la implementación MicroMIPS multiciclo del capítulo 14 se realiza en forma directa y no requiere ver más allá del reconocimiento del estado de control adecuado en el que se deben verificar las solicitudes de interrupción. Los detalles se dejan como ejercicio.

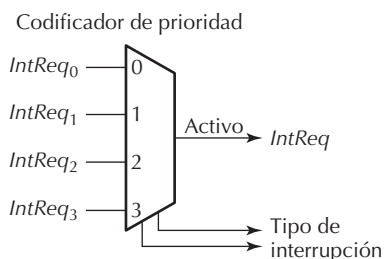


Figura 24.5 Uso de un codificador de prioridad para producir una señal de solicitud de interrupción sencilla y un tipo de interrupción a partir de múltiples solicitudes.

Sin embargo, agregar una capacidad de interrupción a los diseños MicroMIPS encauzados de los capítulos 15 y 16 requiere más cuidado. Los conflictos involucrados son idénticos a los de las excepciones, según se discutió en la sección 16.6. Para el caso de una *pipeline* lineal simple, similar a las de las figuras 15.9 y 15.11, se puede limpiar al permitir que todas las instrucciones, actualmente en varias etapas del encauce, corran hasta su conclusión; eso puede requerir que se inserten $q - 1$ burbujas en un encauce de q etapas. Por ejemplo, si una de las instrucciones de la *pipeline* encuentra un fallo de caché de datos, ocurrirá un largo atasco de *pipeline* mientras se accede a la memoria principal. Para el caso de memoria virtual, existe la posibilidad de una falla de página con su latencia todavía mayor, ello no puede ser admisible para las interrupciones que requieren un tiempo de respuesta corto. Una alternativa es anular todas las instrucciones parcialmente completadas que todavía no afectan memoria de datos o contenidos de registro, mientras se tiene cuidado de que, como resultado, no se introducen inconsistencias. Por ejemplo, en la ruta de datos encauzada de la figura 15.11, varias instrucciones afectan la caché de datos y los contenidos de registro en diferentes etapas de la *pipeline*. Si una instrucción ya tuvo un efecto irreversible, dicha instrucción, y todas las instrucciones por delante de ella en la *pipeline*, deben correr hasta su conclusión como consecuencia de evitar inconsistencias.

Con una *pipeline* bifurcada o conflicto de instrucción fuera de orden (figura 16.8), se puede o limpiar (*flush*) la *pipeline* al permitir que todas las instrucciones corran hasta su conclusión, o se descarten todas las instrucciones parcialmente ejecutadas cuyos resultados todavía no se hayan comprometido. Esto último requiere que el siguiente valor PC asociado con la última instrucción comprometida se mantenga en el retiro y comprometa parte de la *pipeline*. El último enfoque desperdicia algún trabajo que se realizó en las instrucciones no comprometidas, pero permite una reacción más rápida ante las interrupciones. Observe que, en un procesador moderno, docenas de instrucciones pueden estar en varias etapas de conclusión, y algunas de ellas encuentran retardos significativos como resultado de razones resaltadas en párrafos precedentes. Por tanto, en virtud de que la lógica de retiro y compromiso ya asegura la consistencia de todas las instrucciones completadas, se deben descartar todas las instrucciones no comprometidas.

■ 24.4 Interrupciones anidadas

Una idea obvia para manipular múltiples dispositivos I/O de alta rapidez o aplicaciones de control sensibles al tiempo consiste en permitir el anidado de interrupciones de modo que una solicitud de interrupción de máxima prioridad pueda atribuirse (*preempt*) el programa que manipula una interrupción de baja prioridad. La ejecución de este último se resumiría cuando la interrupción de alta prioridad se haya atendido. Sin embargo, las interrupciones anidadas son más difíciles de organizar que las solicitudes de procedimiento anidado; lo último se manipula correctamente con la ayuda de una pila para almacenamiento de variables locales y salvamento de direcciones de retorno. Las dificultades al lidiar con interrupciones anidadas surgen de su temporización impredecible. Un programador que escribe un procedimiento que pide otro de éstos asegura que la segunda petición no se haga hasta después de que todos los pasos previos se hayan completado (figura 6.2). Algo similar no se sostiene para las interrupciones: una interrupción de alta prioridad puede ocurrir después de una prioridad menos importante, quizá después de que sólo una sola instrucción se haya ejecutado. Por tanto, es importante que interrupciones posteriores se deshabiliten hasta que el manipulador de interrupciones haya tenido oportunidad de salvar suficiente información para permitir que la manipulación de la interrupción de prioridad más baja se resume luego que la prioridad alta se haya atendido.

La necesidad de las interrupciones anidadas es evidente a partir del ejemplo 24.1, se relaciona con las garantías de tiempo de respuesta de CPU requerido para ciertos tipos de interrupción. Si el esquema de interrupción de cuatro niveles que se muestra en la figura 24.5 se utilizase para la aplicación de con-

trol del ejemplo 24.1, una interrupción con cualquier nivel de prioridad tendría un tiempo de respuesta de 100 ms, pues la atención de una interrupción del acondicionador de aire pudo iniciarse al llegar la solicitud de interrupción de mayor prioridad. Esto último es inaceptable. Como otro ejemplo, un *drive* de disco que lee a 3 MB/s y transfiere datos al CPU en porciones de 4 B (ejemplo 22.7) requiere que su solicitud de interrupción se atienda dentro de 0.75 μ s. Si hubiera algún tipo de interrupción cuya atención tarda más de 0.75 μ s, y si durante la atención de una interrupción no se acepta otra similar, entonces el hecho de dar al controlador de disco la máxima prioridad no resolvería el problema; se requiere algún tipo de capacidad de atribución.

Considere ahora los pasos que sigue el CPU para atender una interrupción, como se enumeran en la sección 22.4. Si supone una implementación no encauzada, las siguientes acciones ocurren en ruta hacia, y durante, la atención de la interrupción.

1. Deshabilitar (enmascarar) todas las interrupciones.
2. Pedir la rutina de atención de interrupción; salvar el PC como en una petición de procedimiento.
3. Salvar el estado del CPU.
4. Salvar información mínima acerca de la interrupción en la pila.
5. Habilitar las interrupciones (o al menos las de máxima prioridad).
6. Identificar la causa de la interrupción y atender la solicitud subyacente.
7. Restaurar el estado del CPU a lo que existía antes de la última interrupción.
8. Regresar de la rutina de atención de interrupción.

Otra interrupción se puede aceptar después del paso que se indica en el paso 5. En consecuencia, mientras más rápido se ejecuten los primeros cinco pasos, más baja será la latencia de peor caso antes de que se acepte una interrupción de máxima prioridad. Por ejemplo, la interrupción de prioridad más alta debe esperar la ejecución de estos cinco pasos en el peor caso. Cualquier interrupción de prioridad más baja esperará el tiempo de atención del peor caso de todas las posibles interrupciones de máxima prioridad, incluidas las nuevas que puedan ocurrir en el curso de atención de alguna de tales interrupciones.

La figura 24.6 muestra la relación de un programa de aplicación con dos interrupciones anidadas. La primera interrupción (prioridad más baja) se detecta después de la conclusión de la instrucción *a*)

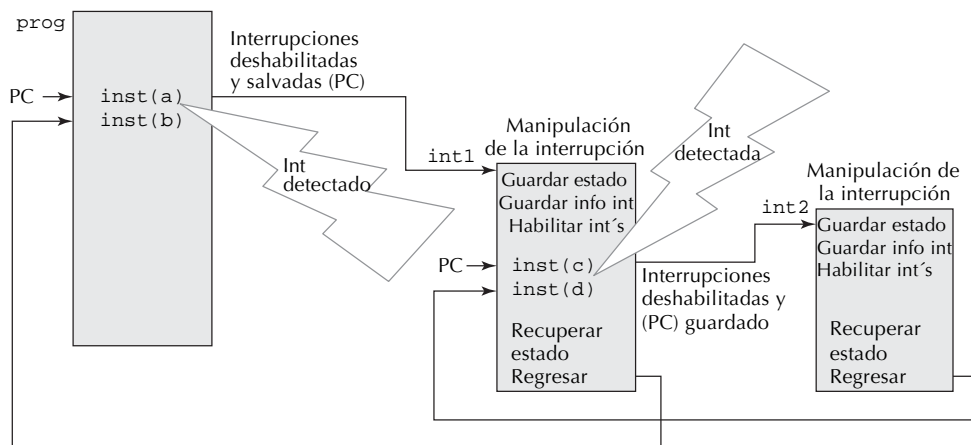


Figura 24.6 Ejemplo de interrupciones anidadas.

y antes de la instrucción *b*) inicia su ejecución dentro del programa de aplicación. A este punto, el PC contiene la dirección de la instrucción *b*), guardando el contenido que permitirá al hardware regresar al punto de la interrupción y continuar ejecutando la aplicación como si nada hubiera pasado entre las instrucciones *a*) y *b*). Todas las interrupciones se deshabilitan (enmascaran) inmediatamente y el control se transfiere automáticamente a un manipulador de interrupción. Luego que el manipulador de interrupción haya salvado el estado del CPU y alguna información mínima acerca de la interrupción, habilita todas las interrupciones de prioridad más alta antes de proceder a manipular la interrupción existente. Si poco después de este punto se detecta una segunda interrupción (prioridad más alta) entre las instrucciones *c*) y *d*), entonces se repite el proceso y se abandona la atención de la interrupción de prioridad más baja en favor de la recién llegada.

■ 24.5 Tipos de conmutación contextual

A la transferencia de control de un programa que corre a un manipulador de interrupción, o desde el manipulador para una interrupción de prioridad baja hacia el de una interrupción de prioridad mayor, se le conoce como *conmutación contextual*. El contexto de un proceso de ejecución consiste del estado de CPU, que incluye contenidos de todos los registros, junto con ciertos elementos de información de gestión de memoria que permiten que el proceso accese a su espacio de dirección. Observe que pedir un procedimiento no se considera una conmutación de contexto porque el procedimiento generalmente opera dentro del mismo contexto que el programa que lo pide; por ejemplo, puede acceder a las variables globales del programa solicitante y pasarle los parámetros a través del archivo de registro o pila. Además de conmutaciones contextuales obligatorias debidas a interrupciones, el sistema operativo puede realizar conmutación contextual voluntaria debido a que se relaciona con el mejoramiento del rendimiento global del sistema. Este tipo de conmutación contextual se realiza dentro del marco de *multiprogramación* o *multitareas*, esos conceptos se explican en esta sección. Un término relacionado, *multihilos*, se explicará en la sección 24.6.

La ejecución de secuencias de instrucciones dentro de un programa se puede retardar por varias razones. Éstas incluyen tanto atascos cortos para eventos como fallos de caché o fallos TLB (capítulos 18 y 20), como esperas mucho más largas para fallas de página y solicitudes I/O. Hasta ahora, en este libro se vieron tales atascos y esperas como contribuyentes a reducir el rendimiento. Éste constituye un enfoque correcto en lo concerniente a la ejecución de una sola tarea secuencial. Sin embargo, así como el tiempo de una persona no se desperdicia realmente si ojea el periódico o lee mensajes de correo electrónico mientras cuelga el teléfono, los recursos de la computadora o del CPU tampoco se desperdician si hay otro programa o tarea en ejecución mientras una tarea encuentra un retardo prolongado.

El uso de multiprogramación para traslapar los periodos de espera de un programa con la ejecución de otros programas, comienza con la compartición de tiempo de los sistemas de cómputo de principios de la década de 1960. En un sistema que comparte tiempo, múltiples programas de usuario residen en la memoria principal de la computadora, ello permite al CPU cambiar entre ellos con relativa facilidad. Para asegurar justicia o parcialidad, a cada programa se le puede asignar una *rebanada de tiempo* fijo en el CPU, y la tarea se hace a un lado en favor de otra al final de su tiempo asignado, incluso si no encuentra esperas. El número de tareas simultáneamente activas en memoria principal representa el *grado de multiprogramación*. Es benéfico tener un alto grado de multiprogramación porque hace más probable que el CPU encuentre trabajo útil incluso en el evento de que muchas tareas encuentran esperas prolongadas. Por otra parte, poner muchas tareas en memoria principal significa que cada una obtiene una asignación de memoria más pequeña, que está en detrimento del rendimiento. La multitarea es esencialmente la misma idea que la multiprogramación, excepto que las tareas activas pueden pertenecer al mismo usuario o incluso pueden representar a diferentes partes (subcálculos) de un solo programa.

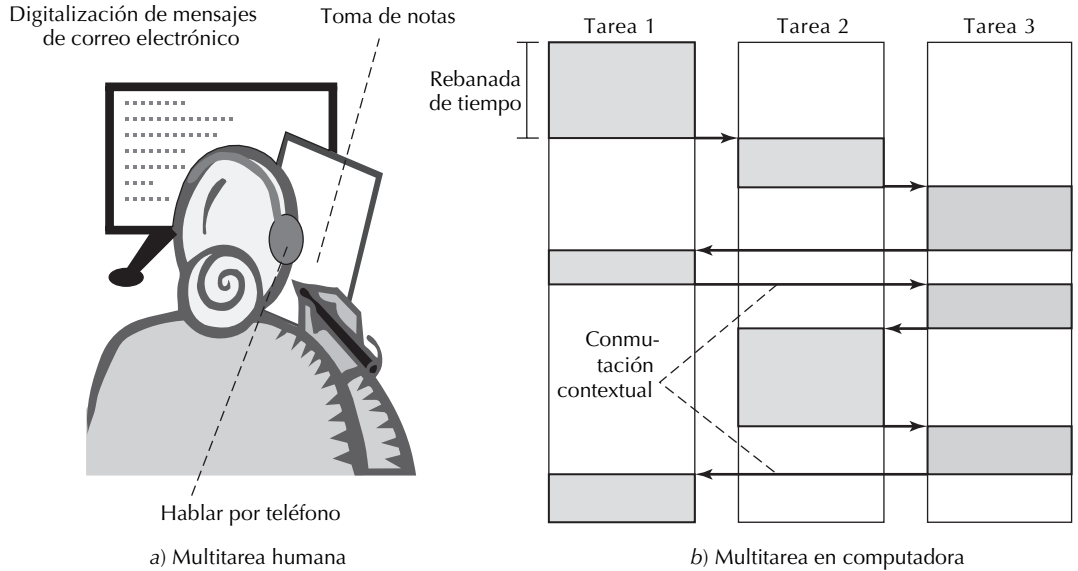


Figura 24.7 Multitarea en humanos y computadoras.

La figura 24.7 muestra multitarea en humanos y computadoras. Como se ve en la figura 24.7b), un CPU puede ejecutar muchas tareas al correr la atención de una a otra a través de la conmutación contextual. El CPU cambia de ida y vuelta entre diferentes tareas a un ritmo rápido que crea la ilusión de procesamiento paralelo, aun cuando, en un momento dado, sólo se ejecute una sola tarea. Para cambiar contexto, se salva el estado del proceso en ejecución en un bloque de control de proceso y se activa un nuevo proceso desde su principio o desde un punto salvado anteriormente. Salvar y restaurar docenas de registros y otra información de control con cada conmutación contextual toma muchos microsegundos que, en cierta forma, es tiempo desperdiciado. Sin embargo, los cientos o incluso miles de ciclos de reloj empleados en una conmutación contextual todavía son significativamente menores que muchos millones de ciclos de reloj requeridos para manipular una falla de página o una operación I/O.

En el contexto anterior, una invocación específica de un programa o una tarea con frecuencia se refieren como *proceso*. Observe que dos procesos distintos pueden ser diferentes invocaciones del mismo programa o tarea. En este sentido, el *multiprocesamiento* se podría usar como un término amplio para multiprogramación o multitarea. Sin embargo, en la literatura acerca de arquitectura de computadoras, el multiprocesamiento tiene un significado técnico que se relaciona con la ejecución concurrente de muchos procesos, en oposición a la aparente concurrencia apenas discutida.

Como consecuencia de que la cabecera de la conmutación contextual no es trivial, la conmutación contextual simple es práctica sólo para usar largos periodos de espera y no se puede usar para recuperar esperas más cortas debido a fallos de caché y similares. Además de la cabecera tangible para salvar y restaurar estados de proceso, la conmutación contextual involucra otros tipos de cabecera que son mucho más difíciles de cuantificar. Por ejemplo, compartir los cachés de instrucción y datos por muchos procesos puede conducir a conflictos y sus acompañantes fallos de caché para cada proceso. De igual modo, cuando los procesos no son completamente independientes uno de otro, la sincronización (para asegurar que las dependencias de datos se cumplen) desperdicia algún tiempo y otros recursos. El siguiente ejemplo captura los efectos de cabecera tangibles en conmutación contextual.

Ejemplo 24.2: Cabecera de interrupciones anidadas Suponga que cada vez que una interrupción de mayor prioridad atribuye una prioridad baja, se incurre en una cabecera de 20 μ s para salvar y restaurar el estado de CPU y otra información pertinente. ¿Esta cabecera es aceptable para la aplicación de sistema de control presentada en el ejemplo 24.1?

Solución: Si ignora la prioridad dada al detector de impacto, porque no genera una interrupción durante operación normal del sistema, existen tres niveles de interrupción. Se mencionan a continuación, con sus tasas de repetición y tiempos de servicio asociados: combustible/encendido (500/s, 1 ms) y pantalla de tablero (800/s, 0.2 ms) tienen la mayor prioridad, temperatura de motor (1/s, 100 ms) está en el nivel medio, y el acondicionador de aire (1/s, 100 ms) tiene la menor prioridad. Incluso, al suponer que la ejecución de los manipuladores de interrupción para temperatura de motor y acondicionamiento de aire se extienden sobre todo un segundo debido a atribuciones repetidas, cada cual se atribuye no más de 1 300 tiempos, para una cabecera total de $2 \times 1\,300 \times 20 \mu s = 52$ ms. Si considera esta cabecera de atribución junto con el tiempo de atención de interrupción total de $500 \times 1 + 1 \times 100 + 800 \times 0.2 + 1 \times 100 = 860$ ms, se observa que todavía habrá un margen de seguridad igual a $1\,000 - 860 - 52 = 88$ ms en cada segundo de tiempo real. De este modo, la cabecera de conmutación contextual es aceptable en este ejemplo de aplicación.

Debido a la importancia de la conmutación contextual en la manipulación eficiente de interrupciones y multitarea, algunas arquitecturas proporcionan recursos de hardware para auxiliar a este propósito. Por ejemplo, suponga que un CPU contiene cuatro archivos de registro idénticos cada uno de los cuales se puede elegir como el archivo de registro activo al establecer una tag de control de dos bits. En consecuencia, la conmutación contextual entre hasta cuatro procesos activos se podría realizar sin salvar o restaurar alguno de los contenidos de registro, al ajustar la tag de control de dos bits. Este es un buen ejemplo de *conmutación contextual de cabecera baja*. Una alternativa menos drástica es proporcionar instrucciones de máquina especiales que salven o restauren todo el archivo de registro en una operación. Es probable que tales instrucciones sean más rápidas que una secuencia de instrucciones que tratan con salvar o restaurar los registros uno a la vez.

■ 24.6 Hilos y multihilos

Como se representa en la figura 24.8, los *hilos* (*threads*) son corrientes de instrucciones (pequeños segmentos de programas) dentro de la misma tarea o programa que se pueden ejecutar concurrentemente con, y durante la mayor parte independientemente de, otros hilos. Como tal, proporcionan una fuente ideal de instrucciones independientes para llenar la *pipeline* de ejecución de CPU y hacer buen uso de las múltiples unidades de función, como las que se muestran en la figura 16.8. A los hilos a veces se les llama *procesos ligeros* porque comparten una gran cantidad y, por tanto, cambiar entre ellos no involucra tanta cabecera como los procesos convencionales (o pesados).

Las arquitecturas que pueden manipular muchos hilos se conocen como *superhiladas*, donde “super” caracteriza el ancho del hilado en forma muy parecida a como *superpipelining* se nombra debido a la profundidad de la *pipeline*. El término *hiperhilado* también se usa para una forma más flexible de multihilo (simultáneo), pero aquí no se tratarán las diferencias sutiles de los dos esquemas. El multihilo divide efectivamente los recursos de un solo procesador en dos o más *procesadores lógicos*, donde cada procesador lógico ejecuta instrucciones desde un solo hilo. La figura 24.9 muestra cómo las instrucciones de tres hilos diferentes se pueden mezclar en las varias etapas de la *pipeline* de ejecución en un procesador de doble emisión.

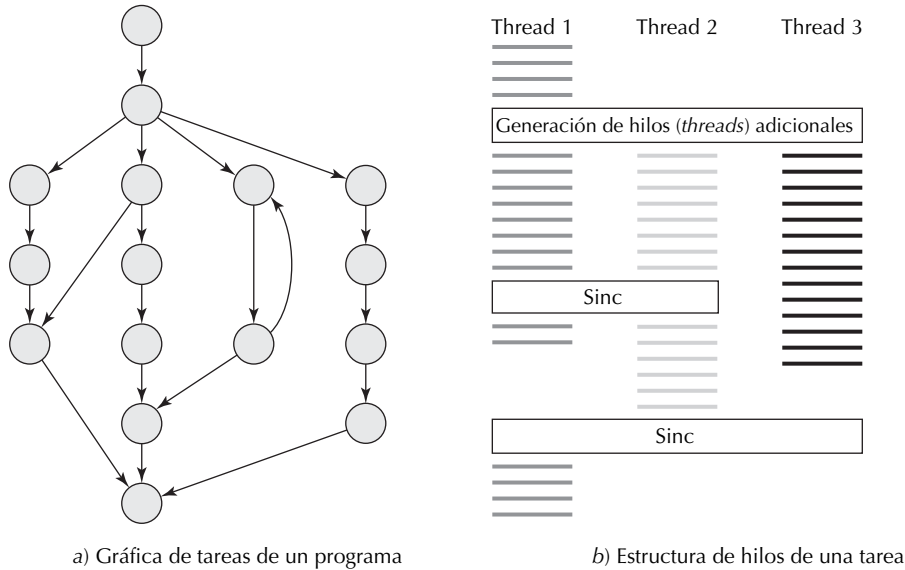


Figura 24.8 Un programa dividido en tareas (subcálculos) o hilos.

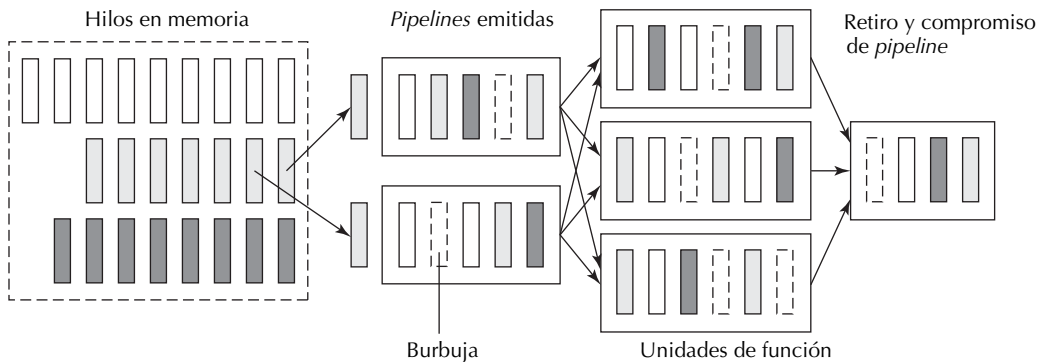


Figura 24.9 Instrucciones de múltiples hilos conforme avanzan en su ruta hacia una *pipeline* de ejecución de un procesador.

Observe que, incluso con multihilos, pueden aparecer algunas burbujas de *pipeline*. Sin embargo, es probable que el número de burbujas sea mucho menor cuando están disponibles para ejecución un gran número de hilos. Construir un programa en forma de multihilos es parecido a liberar al CPU en gran medida de la carga de descubrir paralelismo en el nivel de instrucción y permitir que múltiples instrucciones se emitan a partir de partes dispares de un programa en vez de estar confinadas a instrucciones cercanas en el flujo de control del programa.

PROBLEMAS

24.1 Uso de temporizadores de intervalo

Una computadora tiene un temporizador de intervalo que se puede fijar a una longitud de tiempo deseada (por decir, 0.1 s), en cuyo final se genera una interrupción.

- Describa cómo usaría tal temporizador para construir un reloj análogo, con el uso de un motor de velocidad gradual para salida. El motor necesita controlar sólo la segunda manecilla del reloj, las manecillas de los minutos y las horas se mueven adecuadamente como resultado.
- ¿Cuáles son las fuentes de imprecisión en el tiempo que muestra el reloj de la parte a)?

24.2 Nociones de manipulación de interrupciones

Considere las interrupciones al comer, en la llamada telefónica y al atender la alerta de correo electrónico que se muestran en la figura 24.1.

- Escribe (en idioma llano) una rutina de manipulación de interrupción para estos eventos. Ponga especial atención a la posibilidad de anidar interrupciones. Establezca sus suposiciones.
- Agregue dos niveles de interrupción más a los dos ya presentes en la figura. Describa las nuevas interrupciones en detalle y establezca por qué son de mayor prioridad que las existentes.
- Discuta brevemente cómo estas interrupciones y sus manipulaciones son diferentes de las que están en la computadora.

24.3 Petición de procedimiento frente a interrupciones

Explique las diferencias entre una petición de procedimiento y la transferencia de control a un manipulador de interrupción. Por ejemplo, ¿por qué en el caso de una petición de procedimiento es adecuado cambiar el contenido del PC (es decir, se permite la conclusión en forma normal de las instrucciones adelante de la petición de procedimiento en la *pipeline*), mientras que para una interrupción ésta se debe limpiar?

24.4 Múltiples líneas de solicitud de interrupción

- Muestre cómo, en la figura 24.5, cada tipo de interrupción se puede enmascarar separadamente.

- Agregue lógica de reconocimiento de interrupción a la figura 24.5. Asegúrese de que, cuando se reconozca la interrupción de mayor prioridad, la postulación continua de una señal de solicitud de interrupción de prioridad más baja no conduce a reconocimiento falso.

24.5 Interrupciones en MicroMIPS multiciclo

La adición de capacidad de interrupción a la implementación MicroMIPS de ciclo sencillo del capítulo 13 se discutió en la sección 24.3. Discuta cómo se puede agregar la misma capacidad al MicroMIPS multiciclo del capítulo 14. *Sugerencia:* Piense en términos de la máquina de estado de control de la figura 14.4.

24.6 Interrupciones anidadas

Tres dispositivos D_1 , D_2 y D_3 están conectados a un bus y usan I/O basado en interrupción. Discuta cada uno de los siguientes escenarios en dos formas, una al suponer el uso de una sola línea de solicitud de interrupción y otra al suponer el uso de dos líneas de solicitud de interrupción con prioridades altas y bajas. En cada caso, especifique cuándo y cómo las interrupciones se habilitan o deshabilitan.

- Las interrupciones no estarán anidadas.
- Las interrupciones para D_1 y D_2 no están anidadas en relación una con otra, pero las interrupciones desde D_3 se deben aceptar en cualquier momento.
- Se permitirán tres niveles de anidado, con D_1 en el extremo bajo y D_3 en el extremo alto del espectro de prioridad.

24.7 Conceptos multitarea

Los humanos realizan gran cantidad de multitareas en la vida diaria. En la figura 24.7a se muestra un ejemplo.

- ¿Qué características de las tareas se requieren para hacerlas sensibles a multitarea por humanos?
- ¿Qué tipos de tarea no son adecuados para multitarea por humanos?
- Relacione sus respuestas con las partes a) y b) para multitarea en una computadora.
- Construye un escenario realista en el que un humano realice cinco o más tareas concurrentemente.

- e) ¿Hay un límite sobre el número máximo de tareas concurrentes que se puedan realizar por un humano?
- f) Relacione su respuesta a las partes d) y e) a la multitarea en una computadora.

24.8 Granularidad de la rebanada de tiempo en multiprogramación

Considere las siguientes suposiciones simplificadoras en un sistema de multiprogramación. Los tiempos de ejecución de tarea están distribuidos uniformemente en el rango $[0, T_{\text{máx}}]$. Cuando una rebanada de tiempo de duración τ se asigna a una tarea, se usa completamente y sin desperdicio (esto implica que cálculo e I/O están traslapados), excepto en la última rebanada de tiempo, cuando la tarea se completa; en promedio, la mitad de esta última rebanada de tiempo se desperdicia. La cabecera de tiempo de conmutación contextual es una constante c y no hay otra cabecera.

- a) Determine el valor óptimo de τ para minimizar el desperdicio, si supone que $T_{\text{máx}}$ es mucho más grande que τ .
- b) ¿Cómo cambia la respuesta a la parte a) si los tiempos de ejecución de tarea están distribuidos uniformemente en $[T_{\text{mín}}, T_{\text{máx}}]$? Establezca sus suposiciones.

24.9 Múltiples niveles de prioridad de interrupción

Siguiendo la introducción del ejemplo 24.1, defina un sistema en tiempo real con múltiples niveles de prioridad de interrupción, donde los requerimientos de tiempo de respuesta se pueden satisfacer sin necesidad de interrupciones anidadas. Luego, caracterice tales sistemas en general.

24.10 Auxiliares de hardware para conmutación contextual

- a) ¿Cuál de las instrucciones “complejas” mencionadas en la tabla 8.1 facilitan la implementación de interrupciones anidadas y por qué?
- b) Considere una arquitectura de conjunto de instrucciones de su elección y mencione todas sus instrucciones de máquina que se hayan proporcionado para hacer más eficiente la conmutación contextual.

24.11 Cabecera de conmutación contextual

En el ejemplo 24.2, ¿cuál es la cabecera de atribución máxima que sería aceptable?

24.12 Control de la frecuencia de sondeo

En el ejemplo 22.4 se afirmó que es necesario interrogar un teclado al menos diez veces por segundo para asegurarse de que no se pierda ninguna presión de tecla del usuario. Con base en lo que aprendió en cada capítulo, ¿cómo se puede asegurar de que el sondeo con frecuencia se hace suficiente, pero no con demasiada frecuencia para desperdiciar el tiempo?

24.13 Mecanismos de interrupción en procesadores reales

Para un microprocesador de su elección, describa el mecanismo de manejo de interrupción y su impacto en la ejecución de instrucción. Ponga especial atención a los siguientes aspectos de interrupciones:

- a) Señalamiento y reconocimiento.
- b) Habilitación y deshabilitación.
- c) Prioridad y anidado.
- d) Instrucciones especiales.
- e) Hardware frente a software.

24.14 Niveles de prioridad de interrupción

Existen muchas similitudes entre la manipulación de múltiples niveles de prioridad de interrupción y arbitraje de bus, como se discutió en la sección 23.4. Para cada uno de los conceptos siguientes en interrupciones o arbitraje de bus, indique si hay o no una contraparte en la otra área. Justifique su respuesta en cada caso.

- a) Prioridad rotatoria.
- b) Encadenamiento margarita.
- c) Arbitraje distribuido.
- d) Enmascaramiento de interrupción.
- e) Anidado de interrupción.

24.15 Interrupción de software

La instrucción `syscall` de MiniMIPS (sección 7.6) permite que un programa que corre se interrumpa a sí mismo y pase el control al sistema operativo. El microprocesador ARM tiene una instrucción similar `SWI` (interrupción de software), donde la atención que se so-

licita está especificada en los ocho bits de orden inferior de la instrucción misma en lugar de a través del contenido de un registro, como en MiniMIPS. Cada uno de los servicios proporcionados por el sistema operativo tienen una rutina asociada en ARM y las direcciones de inicio de estas rutinas se almacenan en una tabla.

- a) Describa el mecanismo de interrupción de ARM e indique cómo SWI encaja en él.
- b) Proporcione un panorama del conjunto de instrucciones de ARM, con enfoque en cualquier característica inusual o única.
- c) Identifique las instrucciones ARM que el sistema operativo puede usar para realizar la tarea de transferir control a la rutina de sistema adecuada.

24.16 Circuito de interrupción multinivel

En este problema considere el diseño de un circuito de interrupción de prioridad multinivel externo al procesa-

dor. Este último tiene una sola solicitud de interrupción y una línea de reconocimiento de interrupción. El circuito a diseñar tiene un registro interno de tres bits que contiene el nivel de prioridad actual: un número de 0 a 7. Ocho líneas de solicitud de interrupción R_0 - R_7 entran al circuito, y R_0 tiene la máxima prioridad. Cuando se postula una señal de solicitud de prioridad mayor que el nivel de tres bits almacenado, se envía al procesador una señal de solicitud de interrupción. Cuando el procesador reconoce la interrupción, el nivel de interrupción se actualiza inmediatamente y se envía una correspondiente señal de reconocimiento (la señal de reconocimiento A_i corresponde a la señal de solicitud R_i).

- a) Presente el diseño del circuito de interrupción de prioridad externo según se describe.
- b) Explique el funcionamiento de su circuito, y ponga especial atención a la forma en la que el registro de nivel de tres bits se actualiza para contener un valor más pequeño o más grande.

REFERENCIAS Y LECTURAS SUGERIDAS

- | | |
|----------|--|
| [Egge97] | Eggers, S. J., J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm y D. M. Tullsen, "Simultaneous Multithreading: A Platform for Next Generation Processors", <i>IEEE Micro</i> , vol. 17, núm. 5, pp. 12-18, septiembre/octubre 1997. |
| [Heur04] | Heuring, V. P., and H. F. Jordan, <i>Computer System Design and Architecture</i> , Prentice Hall, 2a. ed., 2004. |
| [Marr02] | Marr, D. T., <i>et al.</i> , "Hyper-Threading Technology Architecture and Microarchitecture", <i>Intel Technology J.</i> , vol. 6, núm. 1, 14 de febrero de 2002. Disponible en: http://www.intel.com/technology/itj/ |
| [Silb02] | Silberschatz, A., P. B. Galvin y G. Gagne, <i>Operating System Concepts</i> , Wiley, 6a. ed., 2002. |
| [Ward90] | Ward, S. A. y R. H. Halstead Jr, <i>Computation Structures</i> , MIT Press, 1990. |

PARTE SIETE

ARQUITECTURAS AVANZADAS

“Durante una década, los profetas han expresado la afirmación de que la organización de una sola computadora alcanzó su límite y que los avances verdaderamente significativos sólo se pueden hacer mediante la interconexión de una multiplicidad de computadoras... [Nuestra meta es demostrar] la continua vitalidad del enfoque de un solo procesador.”

Gene Amdahl, 1967

“Las máquinas con partes intercambiables ahora se pueden construir con gran economía de esfuerzo... El mundo ha llegado a una era de complejos dispositivos baratos de gran confiabilidad; seguro que algo vendrá de ello.”

Vannevar Bush, Cómo podemos pensar, 1945

TEMAS DE ESTA PARTE

- 25. Hacia un mayor rendimiento
- 26. Procesamiento vectorial y matricial
- 27. Multiprocesamiento de memoria compartida
- 28. Multicomputación distribuida

Al haber cubierto los fundamentos de la arquitectura de sistemas de cómputo y algunos métodos ampliamente usados para el mejoramiento del rendimiento, ahora se está listo para observar técnicas de diseño más avanzadas que están en uso actual o que quizá encontrarán su ruta desde los centros de supercómputo y laboratorios de investigación hacia las máquinas domésticas en el futuro cercano. Muchas formas de procesamiento concurrente ya están en uso incluso en las computadoras de bajo perfil. Cualquier sistema con periféricos o interfaz de red ya tiene múltiples procesadores cooperando. Las estaciones de trabajo con dos o más CPU han estado en el mercado desde hace tiempo, y ahora surgen las computadoras personales con múltiples CPU. El procesamiento paralelo se ve ampliamente como la clave para superar los límites de rendimiento y confiabilidad dictados por las leyes físicas y las restricciones en los procesos de fabricación para circuitos integrados.

Esta parte comienza con una discusión de los métodos usados en los uniprosesadores avanzados, y culmina con un panorama de los procesamiento vectorial y concurrente, en el capítulo 25. Este capítulo se puede ver con la sección respectiva que concluye el libro para los lectores que no están interesados en un estudio más detallado del procesamiento vectorial y paralelo/distribuido. Con el propósito de redondear la cobertura hay tres capítulos acerca de procesadores vectoriales y matriciales,

que forman las categorías más antiguas de las computadoras de alto rendimiento, los sistemas multiprocesador con procesadores firmemente acoplados compartiendo un espacio de dirección común, y los sistemas de multicomputadoras con nodos holgadamente acoplados que usualmente se comunican a través de paso de mensajes.

■ CAPÍTULO 25

HACIA UN MAYOR RENDIMIENTO

“La dificultad más constante para diseñar el motor surgió del deseo por reducir el tiempo en el que se ejecutan los cálculos al más corto posible.”

Charles Babbage, De los poderes matemáticos del motor que calcula

“Regla 8: El desarrollo de un algoritmo rápido es lento.”

Arnold Schönhage, 1994

TEMAS DEL CAPÍTULO

- 25.1** Tendencias de desarrollo pasadas y actuales
- 25.2** Extensiones ISA impulsadas por rendimiento
- 25.3** Paralelismo a nivel de instrucción
- 25.4** Especulación y predicción del valor
- 25.5** Aceleradores de hardware de propósito especial
- 25.6** Procesamientos vectorial, matricial y paralelo

Junto con los extensos esfuerzos por diseñar conjuntos de instrucciones y organizaciones de hardware que corran a mayores tasas de reloj, los diseñadores han desarrollado otras estrategias para mejorar el rendimiento de cómputo. En este capítulo se revisan cuatro clases de tales métodos: extensiones de conjunto de instrucciones, paralelismo a nivel de instrucción, ejecución especulativa y uso de aceleradores de hardware de propósito especial. Luego se muestra cómo la realización en hardware de ciclos de programa en supercomputadoras vectoriales y procesamiento concurrente en sistemas paralelos y distribuidos permiten ir más allá del rendimiento que es posible con un procesador SISD (*single-instruction-stream, single-data-stream* = flujo sencillo de instrucciones, flujo sencillo de datos), que ha sido el foco hasta el momento. Los procesamientos vectorial y paralelo se cubren con mayor detalle en los capítulos 26 al 28.

■ 25.1 Tendencias de desarrollo pasadas y actuales

El rendimiento de las computadoras creció por un factor cercano a 10^4 durante las dos últimas décadas del siglo xx, en la medida en que el nivel de rendimiento esperado de una supercomputadora avanzada, voluminosa y costosa en 1980 ahora está disponible en las computadoras de escritorio o incluso laptop.

En términos aproximados, el estado del poder de cómputo disponible en el amanecer del siglo xxi se puede resumir como:

- GFLOPS en escritorio.
- TFLOPS en centros de supercomputadoras.
- PFLOPS en las mesas de diseño.

El factor de 10^4 en el rendimiento mejorado, que se muestra en concordancia con la ley de Moore en la figura 3.10, vino de dos fuentes complementarias. Un factor de 10^2 , o alrededor de 25% por año, se atribuye a avances tecnológicos que continuamente permitieron circuitos más densos y rápidos. En otras palabras, si se construyese una computadora personal de comienzos de la década de 1980 con la tecnología actual, sin cambio arquitectónico sustancial, quizá tendría las mismas aplicaciones casi 100 veces más rápido. El restante factor de 100 se debe a mejoras arquitectónicas de una vez. A esto último se le conoce como “una vez” porque es improbable que las mejoras en rendimiento similares se puedan sustentar al refinar y extender estos métodos. Las ideas clave son las siguientes:

Método arquitectónico establecido	Factor de mejora
1. <i>Pipelining</i> (y <i>superpipelining</i>)	3–8
2. Memoria caché, niveles 2-3	2–5
3. RISC e ideas relacionadas	2–3
4. Múltiple emisión de instrucciones (superescalar)	2–3
5. Extensiones ISA (por ejemplo, para multimedia)	1–3

Si se mira hacia adelante, para 2020 se esperan mejoras comparables o incluso mayores en relación con las dos últimas décadas. Combinadas con otro factor de 100 en el aumento de rendimiento debido a avances tecnológicos, las siguientes tendencias arquitectónicas pueden contribuir a equiparar la ganancia de rendimiento antes mencionada en las próximas dos décadas. La mayoría de estos métodos no son tan nuevos; lo que es nuevo es su uso en máquinas disponibles producidas a bajo costo en lugar que en prototipos exclusivos de laboratorio, *mainframes* costosos y supercomputadoras de bajo volumen.

Tendencia arquitectónica más nueva	Factor de mejora
6. Multihilos (super, hiper)	¿2-5?
7. Especulación y predicción de valor	¿2-3?
8. Aceleración de hardware	¿2-10?
9. Procesamiento vectorial y matricial	¿2-10?
10. Computación paralela/distribuida	¿2-1000?

Sin embargo, observe que la mayoría de estos son métodos con ámbitos limitados y pueden no contribuir a la mejora en rendimiento en todos los dominios de aplicación. A pesar de esta nota precautoria, no es muy optimista predecir la disponibilidad de supercomputadoras EFLOPS ($\text{exa} = 10^{18}$) y máquinas de escritorio TFLOPS hacia 2020. En el resto de este capítulo se discutirán los métodos arquitectónicos 5-8 de la lista y se proporcionará una breve introducción a los elementos 9 y 10. Los procesamientos vectorial y matricial se cubrirán con mayor detalle en el capítulo 26. Los capítulos 27 y 28 se dedican a multiprocesamiento y multicomputación, respectivamente, como enfoques diferentes a la computación paralela y distribuida.

Cada vez más se vuelve importante el rendimiento por unidad de área de chip o unidad de energía, en lugar de rendimiento absoluto. Éste, de hecho, es el dominio en el que yace la mayoría de los desafíos arquitectónicos futuros. Los modernos procesadores de alto rendimiento son bastantes intensos en área y consumen decenas de watts de energía. La figura 25.1 muestra la tendencia del consumo de energía en los procesadores de propósito general y alto perfil y procesadores de señal digital (DSP, por sus siglas

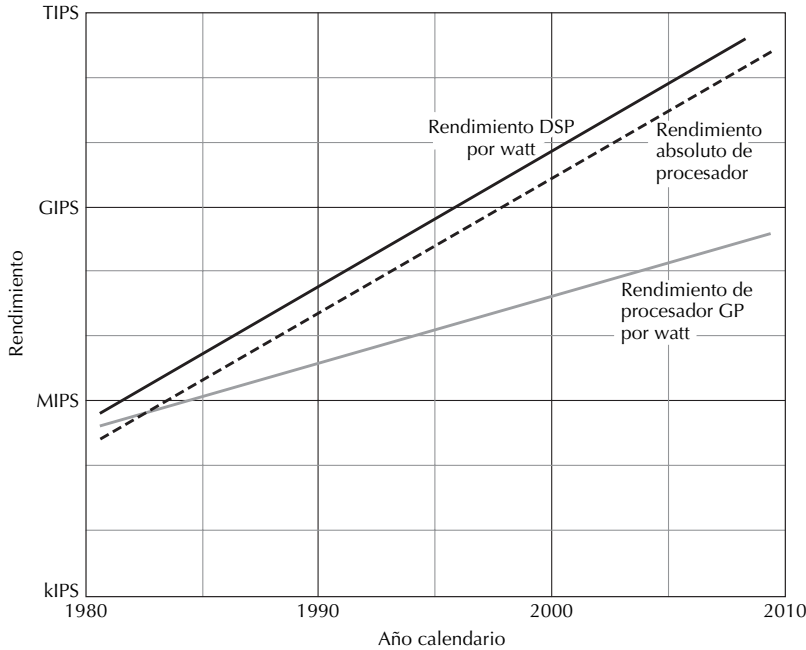


Figura 25.1 Tendencia en consumo de energía para cada MIPS de potencia computacional en procesadores de propósito general y DSP.

en inglés) que están optimizados para usarse en sistemas incrustados y que, por tanto, están diseñados con atención especial a requisitos de energía. Actualmente muchos procesadores de propósito general están implementados en versiones móviles especiales que operan a bajo voltaje y consumen mucho menos potencia que sus contrapartes estándar. Esta tendencia hacia implementaciones de baja potencia continuará como resultado tanto de confianza creciente en los dispositivos operados por batería como en lo deseable de deshacerse de las voluminosas fuentes de poder y equipos de enfriamiento.

Numerosos métodos están disponibles, o en estudio, para la reducción de energía en los procesadores, memorias y dispositivos I/O. Con frecuencia se les categoriza como métodos de diseño de baja potencia, aunque diseño a baja energía representa una designación más adecuada [Mudg01]. Algunos de los métodos que se han aplicado en los niveles de circuito y lógico incluyen los siguientes:

Puertas (*gating*) de reloj: apagar las señales de reloj que van a partes no utilizadas de un chip.

Diseño asíncrono: descartar el cronometrado y el pretendido desperdicio de energía.

Reducción de voltaje: la operación de voltaje bajo es más lenta, los buses usan mucho menos energía.

Adicionalmente, se puede hacer mucho en el nivel arquitectónico, e incluso en programación y esquemas de compilación, para lograr economía de energía. En el aspecto arquitectónico, la memoria y los buses son grandes consumidores de energía. Por tanto, se están adoptando las organizaciones de memoria de baja energía (con partes de memoria desactivadas o puestas a “dormir” cuando no se usan) y protocolos de transferencia especiales que reducen el número de transiciones de señal en los buses del sistema. En algunos diseños móviles especiales, al sistema operativo se le permite escalar dinámicamente el voltaje o la frecuencia de reloj con base en requisitos de la aplicación. Por ejemplo, el procesador Intel XScale puede correr de 100 MHz a 1 GHz, consumiendo menos de 30-40 mW de potencia en el extremo bajo de la frecuencia de reloj.

La reducción en los requisitos del área de chip está motivada, en parte, por el deseo de encajar más funcionalidad en un solo chip VLSI. A principios de la década de 1980, colocar un procesador com-

pleto en un solo chip fue una tarea desafiante que con frecuencia necesitó comprometer el rendimiento. Ahora, un CPU completo (incluido hardware de punto flotante) ocupa sólo una pequeña fracción de un chip, y el área restante de éste se dedica a memorias caché y a algunas unidades auxiliares. Eventualmente se quiere incorporar todo el procesamiento requerido, almacenamiento y unidades de interfaz en el mismo chip, construido con lo que se conoce como *sistema en un chip*. Combinado con avances en entrada (reconocimiento de voz) y salida (micropantallas incrustadas en anteojos), tales computadoras de un solo chip permitirán la construcción de pequeñas computadoras, casi invisibles, que se pueden llevar guardadas en alguna parte del cuerpo.

Entre los requisitos de energía y área de chip existe una interesante negociación. Como consecuencia de que la potencia es proporcional al cuadrado del voltaje, es posible reducir el consumo de energía a través de procesamiento paralelo: dos procesadores de bajo voltaje pueden ofrecer el mismo rendimiento que un solo procesador de alto voltaje a menor costo de energía. Ésta es una de las fuerzas motrices detrás del desarrollo de los *chip multiprocesadores*. Otra parte de esta negociación está en los teléfonos celulares avanzados que usan dos procesadores: uno de baja potencia, pero computacionalmente débil, un procesador de propósito general y otro de señal digital. Un procesador quizá tendría que usar menos área, pero no sería tan eficiente en energía. Una extensión natural de este enfoque es colocar múltiples procesadores (o “núcleos”, *cores*) de baja potencia y bajo rendimiento al otro extremo en el mismo chip y activar la unidad apropiada dependiendo de los requerimientos de rendimiento [Kuma03]. Si estos núcleos se pudieran basar en la misma arquitectura de conjunto de instrucciones (ISA), resultaría significativa simplificación en software y aplicaciones en relación con usar diversos procesadores especializados.

■ 25.2 Extensiones ISA impulsadas por rendimiento

Como se mencionó al comienzo de la sección 8.4, las arquitecturas de conjunto de instrucciones de los procesadores modernos han evolucionado durante muchos años en el contexto de consideraciones de ingeniería y económicas. Con el tiempo, los dominios de aplicación expandidos han conducido a la introducción de nuevas instrucciones en ISA. Los ejemplos de instrucciones que se han introducido para mejorar el rendimiento incluyen:

Multiply-add (multiplicar-sumar): combinar dos operaciones en una instrucción.

Multiply-accumulate (multiplicar-acumular): calcular la suma de productos con mínimo error de redondeo.

Condicional copying (copiado condicional): para evitar algunas bifurcaciones y sus penalizaciones de rendimiento.

Como regla, las funcionalidades necesarias sólo en contextos de aplicación específicos se incorporan en procesadores de propósito especial y son menos atractivos para la inclusión en procesadores de propósito general, dados los posibles efectos adversos en la complejidad del dispositivo y, por tanto, su rendimiento.

En los inicios de la década de 1990, especialmente después de la introducción de la World Wide Web, ocurrió un notable corrimiento en las aplicaciones de computadora. Como resultado, audio, video interactivo, gráficos y animación (que de manera colectiva se denominan *multimedia*) ensombrecieron el número tradicional de trituración (*crunching*) y manipulación de datos como la carga de trabajo dominante para las computadoras de propósito general. Por tanto, se introdujeron varias extensiones arquitectónicas y argumentaciones de conjunto de instrucciones para manipulación más eficiente de la nueva carga de trabajo. Desafortunadamente, el deseo de preservar la compatibilidad retrospectiva y reducir el costo de modificaciones condujo a comprometer, lo que evitó la materialización de los beneficios completos de tales extensiones. No obstante, la idea de introducir instrucciones especiales

para acelerar ciertas operaciones usadas frecuentemente es lo importante como para garantizar un breve repaso en el resto de esta sección.

Las aplicaciones multimedia apenas aludidas se caracterizan por operaciones repetitivas en operando subpalabra. Por “subpalabra” se da a entender operandos que tienen anchos menores de 32 bits. Los ejemplos incluyen atributos de píxel de ocho bits y muestras de audio de 16 bits. Aun cuando una ALU orientada por palabra puede realizar operaciones aritméticas y lógicas sobre operandos más cortos, hacerlo parece desperdiciar tanto hardware como tiempo. Uno de los primeros intentos mejor conocidos para sacar ventaja de esta característica de aplicación para impulsar el rendimiento fue el MMX (multimedia extensión) de Intel, diseñado para el procesador Pentium. MMX consistía de agregar 57 nuevas instrucciones a la ISA 80x86 para manipular muchas operaciones aritméticas subpalabra en una sola instrucción. Las instrucciones parecidas a MMX que permiten paralelismo subpalabra ahora son más o menos estándar en muchos procesadores.

Para entender las instrucciones MMX de Intel, considere un archivo de registro con algunos registros de 64 bits. En realidad, MMX usa ocho registros de punto flotante que duplican el archivo de registro entero MMX, pero esto es sólo una elección de diseño que ahorra costo. Más registros habrían aumentado el impacto de rendimiento de MMX. Un registro MMX de 64 bits se puede ver como conteniendo un solo entero de 64 bits o un vector de dos, cuatro u ocho operandos enteros progresivamente más estrechos. Para cada una de estas longitudes de vector se proporcionan varias instrucciones vectoriales, lógicas, de comparación y reordenamiento. La tabla 25.1 menciona las clases de instrucción y sus efectos. La columna “Vector” indica el número de elementos vectoriales independientes que se pueden especificar en el operando de 64 bits.

■ **TABLA 25.1** Instrucciones Intel MMX.

Clase	Instrucción	Vector	Tipo op	Función o resultados
Copiar	Register copy		32 bits	Registro entero \leftrightarrow registro MMX
	Parallel pack	4, 2	Saturate	Convierte a elementos más estrechos
	Parallel unpack low	8, 4, 2		Une mitades inferiores de dos vectores
	Parallel unpack high	8, 4, 2		Une mitades superiores de dos vectores
Aritmética	Parallel add	8, 4, 2	Wrap/Saturate ¹	Suma; inhibe acarreo en fronteras
	Parallel subtract	8, 4, 2	Wrap/Saturate ¹	Resta con inhibición de acarreo
	Parallel multiply low	4		Multiplica, conserva las cuatro mitades inferiores
	Parallel multiply high	4		Multiplica, conserva las cuatro mitades superiores
	Parallel multiply-add	4		Multiplica, suma productos adyacentes ²
	Parallel compare equal	8, 4, 2		Todos 1 donde igual, de otro modo todos 0
Corrimiento	Parallel compare greater	8, 4, 2		Todos 1 donde mayor, de otro modo todos 0
	Parallel left shift logical	4, 2, 1		Corrimiento izquierdo, respeta límites
	Parallel right shift logical	4, 2, 1		Corrimiento derecho, respeta límites
	Parallel right shift arith	4, 2		Corrimiento aritmético dentro de cada (media)palabra
Lógica	Parallel AND	1	Bitwise	$\text{dest} \leftarrow (\text{src1}) \wedge (\text{src2})$
	Parallel ANDNOT	1	Bitwise	$\text{dest} \leftarrow (\text{src1}) \wedge (\text{src2})'$
	Parallel OR	1	Bitwise	$\text{dest} \leftarrow (\text{src1}) \vee (\text{src2})$
	Parallel XOR	1	Bitwise	$\text{dest} \leftarrow (\text{src1}) \oplus (\text{src2})$
Acceso a memoria	Parallel load MMX reg		32 o 64 bits	Dirección dada en registro entero
	Parallel store MMX reg		32 o 64 bit	Dirección dada en registro entero
Control	Empty FP tag bits			Requerido para compatibilidad ³

¹ Wrap significa destilar la salida de acarreo; la saturación puede ser con o sin signo.

² Cuatro multiplicaciones de 16 bits, cuatro resultados intermedios de 32 bits, dos resultados finales de 32 bits.

³ Los bits de etiqueta de punto flotante ayudan para la conmutación contextual más rápida, entre otras funciones.

Las instrucciones *parallel pack* (empaquetado paralelo) y *unpack* (desempaquetar) son bastante interesantes y útiles. El desempaquetado permite que los elementos vectoriales se extiendan al siguiente ancho más grande (de byte a media palabra, de media palabra a palabra, de palabra a doble palabra) para permitir computación de resultados intermedios con mayor precisión. Las operaciones de desempaquetado se pueden especificar como:

8-vectores, bajo:	xxxxabcd, xxxxefgh → aebfcgdh
8-vectores, alto:	abcdxxxx, efghxxxx → aebfcgdh
4-vectores, bajo:	xxab, xxcd → acbd
4-vectores, alto:	abxx, cdxx → acbd
2-vectores, bajo:	xa, xb → ab
2-vectores, alto:	ax, bx → ab

donde *i*-vectores significa un vector de longitud *i* y las letras representan elementos vectoriales en los operandos (a la izquierda de la flecha) y resultados (a la derecha). Como ejemplo, cuando el primer vector es todo 0, esta operación efectivamente duplica el ancho de los elementos inferior o superior del segundo operando a través de extensión 0 (por ejemplo, 0000, xxcd → 0c0d). El empaquetado realiza la conversión inversa en que permite el retorno de los resultados al ancho original:

4-vector:	a ₁ a ₀ b ₁ b ₀ c ₁ c ₀ d ₁ d ₀ → 0000abcd
2-vector:	a ₁ a ₀ b ₁ b ₀ → 00ab

Por ejemplo, al filtrar valores de píxel, puede ser necesario multiplicar por los correspondientes coeficientes de filtro; usualmente, esto último conduce a desbordamiento si los elementos no se desempacan primero.

Una característica clave de MMX es la capacidad de *aritmética de saturación*. Con aritmética ordinaria sin signo, el desbordamiento hace que el resultado aparente (después de destilar el bit de salida de acarreo) se vuelva más pequeño que cualquier operando. Lo anterior se conoce como *aritmética cíclica* (*wrapped arithmetic*). Cuando se aplica a aritmética sobre atributos de píxel, este tipo de reciclado puede conducir a anomalías como píxeles brillantes dentro de regiones que se supone deben ser bastante oscuros. Con aritmética de saturación, los resultados que exceden el máximo valor representable se fuerzan a dicho valor máximo. Por ejemplo, se puede construir un sumador de saturación sin signo a partir de un sumador ordinario y un multiplexor en la salida que elija entre el resultado del sumador y una constante, dependiendo del bit de salida de acarreo. Para resultados con signo, la aritmética de saturación se puede definir en forma análoga, y el uso del valor más positivo o negativo depende de la dirección de desbordamiento.

La aritmética con paralelismo subpalabra requiere modificar la ALU para trata palabras de 64 bits en una diversidad de formas, dependiendo de la longitud del vector. Para suma cíclica, esta capacidad es fácil de proporcionar y viene de manera gratuita (al deshabilitar los acarreo en las fronteras subpalabra). La suma de saturación requiere detección de desbordamiento dentro de subpalabras y elegir la salida de sumador o una constante como resultado de operación dentro de dicha subpalabra. La multiplicación es ligeramente más difícil, pero todavía bastante efectiva en costo en términos de implementación de circuito (los detalles se dejan como ejercicio). Los efectos de las instrucciones MMX multiplicación paralela y multiplicar-sumar paralela se muestran en la figura 25.2. Las instrucciones de comparación paralela se ilustran en la figura 25.3.

Observe que MMX trata exclusivamente con valores enteros. Una capacidad similar se agregó en una extensión subsecuente a los procesadores Intel para proporcionar aceleraciones similares con operandos punto flotante de 32 o 64 bits, empaquetados dentro de cuatripalabras de 128 bits en registros [Thak99]. Esta última capacidad se conoce como *streaming SIMD extension* (SSE, extensión de transmisión SIMD); en la sección 25.6 se explicará el significado de SIMD.

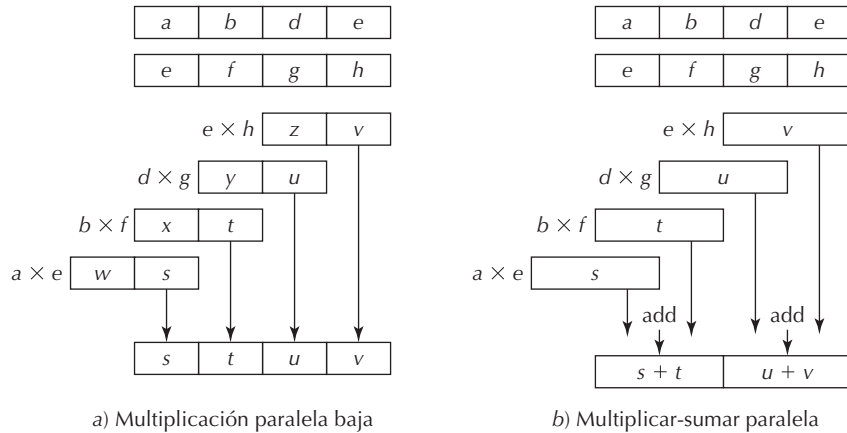


Figura 25.2 Multiplicación paralela y multiplicar-sumar en MMX.

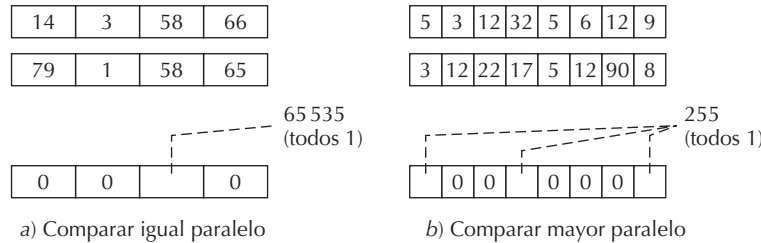


Figura 25.3 Comparaciones paralelas en MMX.

25.3 Paralelismo a nivel de instrucción

El *paralelismo a nivel instrucción* (ILP, por sus siglas en inglés) se refiere a la colección de técnicas que permiten que más de una instrucción se ejecute a la vez. No hay procesador moderno que no tenga alguna forma de paralelismo a nivel instrucción. El *pipelining* es el método ILP más usado. Aquél y su impacto sobre el rendimiento se estudiaron en los capítulos 15 y 16. La emisión de instrucciones múltiples, o procesamiento superescalar, se discutió brevemente en la sección 16.5. Ambos métodos están inherentemente limitados en la mejora de rendimiento que ofrecen. Las limitaciones del *pipelining* se discutieron en la sección 15.3 en relación con el impacto de la cabecera sobre la aceleración (figura 15.8), y en el capítulo 16 en relación con las dependencias de datos y control.

El limitado beneficio de rendimiento de la emisión de instrucciones múltiples surge de las dificultades en la identificación de instrucciones independientes y luego su ejecución ante las contenciones de recurso. La identificación de instrucciones ejecutables de manera independiente usualmente ocurre dentro de una ventana de instrucciones bastante estrecha en una etapa de preprocesamiento. Ensachar esta ventana puede aumentar el número de instrucciones independientes para ejecución paralela, pero también puede conducir a retornos disminuidos porque produce ineficiencias de los tipos siguientes:

1. Frenado de la lógica de emisión de instrucción debido a extrema complejidad de circuito
2. Desperdicio de recursos en instrucciones que no se ejecutarán debido a bifurcaciones

La figura 25.4a muestra el paralelismo en el nivel de instrucción disponible bajo las condiciones ideales de no limitación de recursos, predicción perfecta de bifurcación, etcétera. Incluso con estas su-

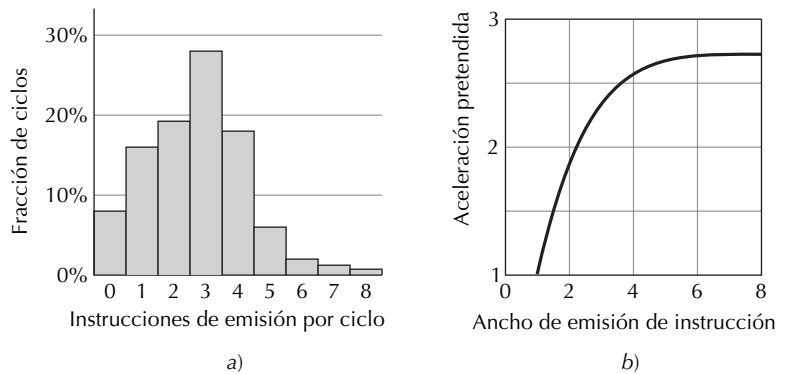


Figura 25.4 Paralelismo a nivel de instrucción disponible y la aceleración debida a emisión de múltiples instrucciones en procesadores superescalares [John91].

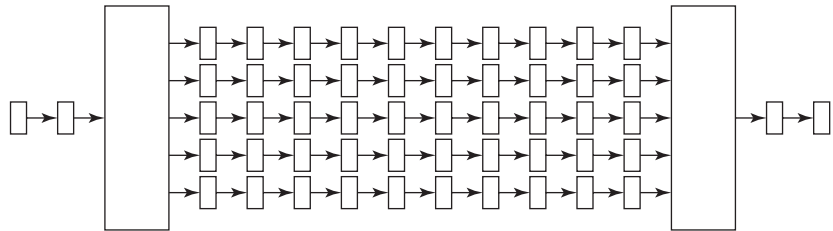


Figura 25.5 Cálculo con paralelismo inherente en el nivel de instrucción.

posiciones optimistas, se pueden emitir cinco o más instrucciones en no más de 10% de los casos. Por tanto, no es de sorprender que la aceleración pretendida a partir de emisión de múltiples instrucciones se colapse más allá del ancho de emisión de cuatro instrucciones (figura 25.4b).

Tanto para *pipelining* como para emisión de múltiples instrucciones, el modelo de programación es secuencial, ejecución de instrucción una a la vez. Este modelo subyacente se fortalece con el hardware, a pesar de concurrencia y ejecución fuera de orden. Este modelo de computación ha servido en el pasado, pero está saliendo de la corriente principal como herramienta para soportar cálculos de alto rendimiento. Para ver por qué, considere el cálculo abstracto que se muestra en la figura 25.5, donde cada una de las cajas pequeñas se visualiza como una instrucción y las flechas denotan la orden de ejecución. Tales cálculos, que comienzan con una fase de arranque inicial, continúan con un número de subcálculos independientes y terminan con una fase de combinación, no son raros en absoluto. El programa que representa el cálculo de la figura 25.5 es probable que tenga la siguiente estructura general: inicializa; realiza el subcálculo 1; realiza el subcálculo 2; . . . ; realiza el subcálculo 5; limpia y cierra. En el programa en lenguaje de máquina resultante, el paralelismo disponible a nivel instrucción, tan claramente visible en la figura 25.5, se pierde por completo y no es probable que se recupere mediante el hardware, que busca paralelismo dentro de una pequeña ventana de instrucciones consecutivas.

Una forma de dar vuelta al problema es usar paralelismo explícito en especificación de instrucción. La filosofía arquitectónica resultante se ha llegado a conocer por medio de dos nombres diferentes, aunque más o menos equivalentes:

Arquitectura de palabra de instrucción muy larga (VLIW, por sus siglas en inglés).

Computación de instrucción explícitamente paralela (EPIC, por sus siglas en inglés).

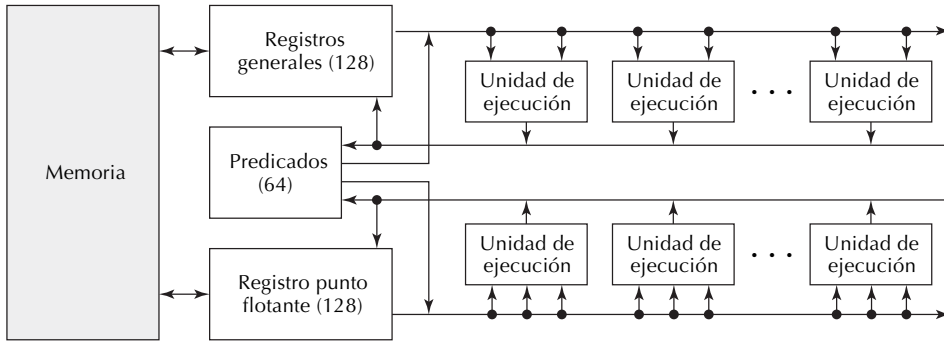


Figura 25.6 Organización de hardware para IA-64: los registros general y de punto flotante tienen 64 bits de ancho; los predicados son registros de un solo bit.

La idea es tener palabras de instrucción muy grandes que contengan múltiples operaciones básicas para ejecución concurrente, ello permite al programador, o con más frecuencia al compilador, especificar cuáles operaciones son capaces de ejecutarse al mismo tiempo. Este es un enfoque mucho más razonable que ocultar la concurrencia en el código de programa secuencial y luego requerir que el hardware lo redescubra. Como consecuencia de que las dependencias y las limitaciones de recursos se pueden tomar en cuenta durante el empaquetado de estas operaciones dentro de las palabras de instrucción largas, el hardware para ejecutar las operaciones se simplifica considerablemente. Liberar al hardware de la difícil tarea de extraer paralelismo y asegurar que las dependencias se satisfacen conduce a economía de circuito, reducción en los requerimientos de energía y rapidez añadida. Además de estas ventajas, es posible mayor paralelismo pues el compilador tiene una visión global del programa, mientras que la lógica de emisión de instrucción de un procesador superescalar busca a través de una ventana estrecha.

La arquitectura de 64 bits de Intel, llamada IA-64 e implementada por primera vez en el procesador Itanium, es ejemplo del enfoque VLIW/EPIC. Como se muestra en la figura 25.6, IA-64 tiene 128 registros generales (64 bits de ancho), 128 de punto flotante (82 bits de ancho) y 64 de predicado (1 bit de ancho). También hay ocho registros de bifurcación (64 bits de ancho), no mostrados en la figura 25.6, que se usan para bifurcación indirecta. La palabra de instrucción larga (*bundle*, *haz*) de IA-64 contiene tres instrucciones tipo RISC (“sílabas”) un poco convencionales, cada una ocupa 41 bits, más una “plantilla” (*template*) de cinco bits que contiene información de calendarización, para un total de 128 bits. La plantilla especifica los tipos de instrucción contenidos en el haz e indica cuáles instrucciones se pueden ejecutar concurrentemente; la concurrencia especificada puede abarcar desde una parte de un haz hasta varios haces. Como se muestra en la figura 25.6, existen muchas unidades de ejecución de varios tipos (entero, multimedia, punto flotante, carga/almacenamiento y bifurcación).

Cada instrucción de 41 bits tiene un *opcode* principal de cuatro bits, una especificación de predicado de seis bits y 31 bits para operandos y modificadores. Por ejemplo, en esta parte se pueden especificar tres registros, donde quedan diez bits para otra información. La especificación de un predicado en cada instrucción permite la ejecución condicional que obvia la necesidad de muchas bifurcaciones condicionales. Los resultados de una instrucción se comprometen sólo si el predicado especificado es 1; de otro modo, se descartan. De esta forma, se pueden ejecutar las instrucciones a lo largo de las partes *then* y *else* de un enunciado condicional, pero sólo un conjunto de resultados comprometidos dependiendo del resultado de la condición. A lo anterior se le conoce como *ejecución de instrucción predicada*. El registro de predicado 0 contiene la constante 1; por tanto, permite ejecución incondicional como un caso especial.

Además de la ejecución predicada, IA-64 usa al menos otros tres métodos para la mejora de rendimiento. Los dos primeros, especulación de control y especulación de datos, se discutirán en la sección 25.4. Ambos métodos especulativos tienen que ver con la realización de una operación de carga de registro adelante en el tiempo, para traslapar la latencia de acceso a memoria con otro trabajo útil. El tercer método es *pipelining de software*, que permite concurrencia entre instrucciones que pertenecen a diferentes iteraciones de ciclo. Por ejemplo, en un ciclo que lee elementos sucesivos de un vector, actualiza cada uno y los almacena en los correspondientes elementos de otro vector, el *pipelining* de software permite que una nueva iteración se ejecute en cada ciclo de reloj, y el compilador asigna diferentes registros para retener los elementos del primer vector conforme se llevan desde memoria. La ejecución ordinaria del ciclo requeriría muchos ciclos de reloj por iteración.

En lugar de VLIW o EPIC, o quizá además de ellos, se puede usar la idea de multihilo (*multithread*, sección 24.6) para exponer al hardware el paralelismo a nivel instrucción disponible. En una arquitectura multihilo, cada hilo puede tener su propio contador de programa y acaso un archivo de registro separado. La lógica de emisión de instrucción superescalar tiene una opción de emitir múltiples instrucciones a partir del mismo hilo o mezclar instrucciones de múltiples hilos. Los tres enfoques multihilo principales son:

Multihilo interpolado.

Multihilo bloqueado.

Multihilo simultáneo.

Con el *multihilo interpolado*, la lógica de emisión de instrucción recoge instrucciones de los diferentes hilos que están disponibles para ejecución en una forma *round-robin*: todos con todos. Este enfoque esencialmente separa las instrucciones del mismo hilo, lo que hace mucho menos probable necesitar una burbuja de *pipeline* por razones de dependencias de datos o control o para desperdiciar tiempo en ejecución especulativa (sección 25.4). El *multihilo bloqueado* esencialmente lidia sólo con un hilo hasta que la ejecución de dicho hilo encuentra un obstáculo (como un fallo de caché). Entonces conmuta a un hilo diferente. La idea es llenar los atascos largos de un hilo con instrucciones útiles de otro hilo. El *multihilo simultáneo* es la estrategia más flexible; permite mezclar y equiparar las instrucciones de diferentes hilos en cada ciclo para hacer mejor uso de los recursos disponibles.

■ 25.4 Especulación y predicción del valor

En la sección 16.4 se discutieron técnicas para predicción de bifurcaciones y su importancia en la reducción de la penalización de rendimiento de las dependencias de control. Dada la ocurrencia relativamente frecuente de las instrucciones tipo *branch* dentro de los programas típicos, no es raro encontrar una instrucción tipo *branch* cada 1-2 ciclos de reloj en un procesador de emisión múltiple de alto rendimiento o VLIW. La predicción simple de bifurcación es inadecuada en este contexto. La *especulación* abarca un conjunto de técnicas para la ejecución parcial de instrucciones antes de que se tenga la certeza de que se visitarán en el flujo de control del programa o de que se hayan obtenido todos los operandos requeridos. Esto último se hace sin comprometer los resultados de tal ejecución de instrucción especulativa. La especulación se puede hacer en hardware o software, ambos enfoques se usan en modernos procesadores de alto rendimiento.

La especulación de software se realiza mediante el compilador, con ayuda del hardware. Dos ejemplos son los mecanismos de especulación de control y especulación de datos del IA-64. La *especulación de control*, que carga datos de memoria antes de que el programa los necesite, representa un esfuerzo por ocultar la latencia de memoria. En términos simples, esto último se realiza mediante la sustitución de cada *load* por dos instrucciones: una *load* especulativa realizada más temprano en el programa y una

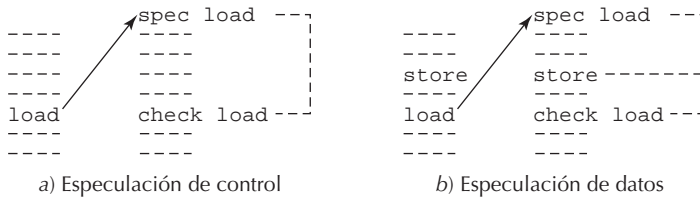


Figura 25.7 Ejemplos de especulación de software en IA-64.

instrucción *checking* donde la instrucción *load* se ubicaba originalmente (figura 25.7a). Al mover una instrucción *load* hacia un punto anterior, hay oportunidad de que se pueda ejecutar una instrucción *load* que en realidad no se encontrará en el flujo de programa normal debido a bifurcación o porque el predicado utilizado resulta ser falso. Para evitar cualquier problema, la *load* especulativa mantiene un registro de cualquier excepción (como violación de protección, falla de página, etc.) sin que en realidad señale la excepción. La instrucción *checking* que sustituye la *load* original realiza la función de señalar una excepción si es adecuado.

La *especulación de datos* en IA-64 permite que una instrucción *load* se mueva a un punto antes de una instrucción *store* más temprana (figura 25.7b). Esto es especulativo si las instrucciones usan direcciones en registros de modo que al momento de compilar no se sabe si las dos instrucciones se refieren a la misma dirección de memoria. Los procesos ordinarios no especulativos respetan el ordenamiento de escritura en memoria frente a los accesos de lectura, para evitar lectura (*fetching*) incorrecta de datos. Esto se debe hacer incluso si sólo hay una ligera oportunidad de que un acceso de escritura o lectura tendrá la misma dirección de memoria. Con la ejecución especulativa, de nuevo se tiene una *load* especulativa que se mueve hacia un punto anterior y una instrucción *checking* que sustituye la *load* original. Cuando la *load* especulativa se realiza, la dirección se graba y verifica contra direcciones subsecuentes para operaciones de escritura de memoria. En caso de equiparación, la instrucción *checking* anula el resultado de la *load* especulativa y cualquier instrucción subsecuente que haya usado el valor cargado.

Otras formas de especulación incluyen instrucciones de emisión de ambas rutas de una instrucción *branch*, con cada grupo predicado en la condición correspondiente. Cuando se conoce el resultado de la condición *branch*, uno u otro grupo de instrucciones se descarta sin comprometer sus resultados. Observe que no es muy obvio que este enfoque conduciría a una ganancia de rendimiento. Las instrucciones de ejecución que se descartan subsecuentemente usan ancho de banda de memoria de instrucción, así como recursos de procesamiento. Para bifurcaciones cuyos resultados se predicen fácilmente, ya sea estáticamente por el compilador o dinámicamente al momento de correrse, este tipo de especulación puede no valer la pena. Sin embargo, para los tipos de bifurcación que resultan de enunciados *si-entonces-de otro modo*, y, por tanto, son muy difíciles de predecir, este enfoque puede conducir a ganancia de rendimiento. El contexto de la especulación también es importante: en una parte de un programa donde esté disponible poco paralelismo en el nivel instrucción, la especulación ofrece mayor ventaja porque usa recursos que de otro modo permanecerían inactivos.

La especulación basada en hardware tiene objetivos similares. Para ver cómo se puede agregar ejecución especulativa a un procesador con conclusión de instrucción fuera-de-orden, primero se ve cómo se puede implementar en hardware la ejecución no especulativa fuera de orden. Se aludió a tal capacidad en la sección 16.5, sin especificar los detalles de la implementación en hardware. Una forma de realizar la ejecución fuera de orden en una máquina de 32 registros como MiniMIPS consiste en proporcionar un número de registros mucho más grande, por decir 128, en hardware. Los últimos *registros microarquitectónicos* o físicos, así llamados para distinguirlos de los *registros arquitectónicos* o lógicos que son visibles a un programa, pueden alojar valores temporales todavía no comprometidos con registros arquitectónicos, y también se pueden usar para remover ciertos tipos de riesgos de datos.

Considere, por ejemplo, una instrucción $add\ \$8 \leftarrow (\$8) + (\$9)$. Como parte del proceso de emisión de instrucción, se renombran los registros, de modo que, en términos de registros físicos, esta instrucción add se puede convertir en $\#52 \leftarrow (\#34 + (\#2))$, donde $\#34$ y $\#12$ son los registros físicos actualmente designados como los registros arquitectónicos $\$8$ y $\$9$, respectivamente, y se elige $\#52$ para contener el resultado. Eventualmente, cuando la instrucción se retira, $\#52$ se convierte en $\$8$ y $\#34$ contiene el valor previo que aparecía en $\$8$. Observe que este valor previo todavía puede estar en uso por otra instrucción cuya conclusión se haya retardado debido a la indisponibilidad de un segundo operando. Cuando este valor antiguo de $\$8$ ya no se necesita más, $\#34$ se regresa a la *pool* de registros libres y se puede reutilizar.

Agregar ejecución especulativa a este esquema ahora es directo. Por ejemplo, cuando las instrucciones se leen (*fetch*) y emiten con base en un resultado de bifurcación predicho, se ejecutan pero sus resultados no se comprometen a registros o memoria hasta que se haya determinado que la predicción fue correcta. Cualquier condición excepcional también se registra junto al resultado, y la excepción toma efecto cuando la instrucción se compromete. Si en una instrucción subsecuente se usa un valor calculado especulativamente, el resultado de aquélla también se designa como especulativo. De este modo, en un momento dado, entre los registros físicos pueden existir múltiples versiones de un registro arquitectónico particular, y cada uno con una denominación adecuada para designar su estatus y las condiciones bajo las que se comprometerá. Como con la especulación de software, la medida de la ejecución especulativa en hardware se determina al considerar la necesidad de remover los atascos de *pipeline* contra el costo en ancho de banda de memoria y los recursos de procesamiento de las instrucciones en ejecución cuyos resultados se descartan subsecuentemente.

La predicción de valor es un método que se puede usar tanto en forma no especulativa como especulativa y, como tal, complementa las técnicas cubiertas hasta el momento en esta sección. La idea es predecir el valor regresado por una instrucción sin, o antes de, ejecutar. Aun cuando pueda parecer que la probabilidad de predecir correctamente uno de 2^{32} resultados posibles de una instrucción en una máquina de 32 bits es bastante pequeña, no es raro lograr 50% de precisión en la predicción en ciertas aplicaciones. Por ejemplo, se puede almacenar los operandos y resultados de instrucciones de división recientes en una pequeña memoria caché conocida como *tabla memo*. Esta tabla, que puede organizarse con mapeo directo basado en unos cuantos bits de cada operando, regresa el resultado de división correcto cuando se realizan operaciones repetitivas (por ejemplo, del tipo que se encuentra en el procesamiento de imagen) sobre elementos de datos que tienden a tener valores agrupados. Este método es útil para instrucciones de gran latencia como la división, que usualmente tarda decenas de ciclos de reloj en la unidad de ejecución y no es benéfica para operaciones más rápidas como la suma y la multiplicación. El mismo enfoque se puede usar para predecir los resultados de una secuencia de instrucciones que realicen un cálculo complicado. Los beneficios de la predicción exitosa son mucho mayores aquí, pero el costo de hardware en términos de necesidades de almacenamiento puede ser prohibitivo.

Observe que, en los ejemplos anteriores, también conocidos como *reuso de instrucción*, no se involucra especulación: los resultados requeridos están en la tabla memo y se regresan, o de otro modo se proporciona una indicación de fallo (muy parecida a las de acceso de instrucción y datos), que causa que la instrucción o secuencia de instrucciones se ejecute en la forma normal. Sin embargo, los beneficios completos de la predicción de valor se materializan cuando la predicción se usa en combinación con especulación o *superespeculación* (otro nombre para especulación agresiva).

Por ejemplo, cuando un valor de operando se necesita para ejecutar una instrucción, ese valor se puede predecir y el cálculo se realizará especulativamente. Entonces el resultado especulativo se convierte en un resultado normal cuando se verifica la exactitud de la predicción. Las estrategias de predicción varían desde el muy simple *predictor de último valor* hasta métodos que toman en cuenta el patrón de resultados previos o la historia del flujo de control. Dado que la ejecución especulativa cuesta

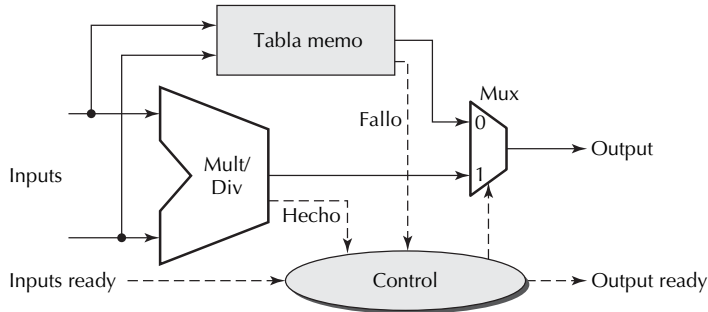


Figura 25.8 Predicción de valor para multiplicación o división a través de tabla memo.

en términos de recursos de cálculo y energía desperdiciados, el enfoque vale la pena si la tasa de equivocaciones se mantiene baja (un pequeño porcentaje). Observe que el enunciado en el párrafo anterior acerca de hasta 50% de resultados predichos no está peleado con una precisión de predicción de, por decir, 98%; simplemente, el procesador no especula en casos que se sabe son difíciles de predecir.

La predicción se puede aplicar a direcciones calculadas, así como a datos. Por ejemplo, con base en un patrón de direcciones previas usadas por una instrucción *load*, es posible predecir la siguiente dirección que se generará e iniciar la carga, aun cuando el cálculo de dirección todavía espere el suministro de un valor de registro necesario. Tal predicción puede ser bastante precisa cuando la memoria se accede a memoria con una tira fija para leer elementos de un arreglo o subarreglo (por decir, una columna dentro de una matriz). De hecho, si se puede determinar casi con certeza que una *load* y una *store* previa tienen diferentes direcciones, la *load* se puede mover adelante de la *store* (como en IA-64, discutido anteriormente) para reducir el “atasco” de datos. Asimismo, si se puede establecer que una *store* y una subsecuente *load* usan la misma dirección, entonces los datos necesarios para la instrucción *load* se pueden adelantar a ella desde la fuente de producción, con lo se obvia la necesidad para un acceso a memoria.

25.5 Aceleradores de hardware de propósito especial

Una alternativa para implementar un procesador demasiado complejo que realice bien cualquier flujo de instrucciones con o sin apoyo de compilador es delegar operaciones especializadas a unidades que estén diseñadas o afinadas para ellas. El resultado global puede ser la ejecución de las operaciones requeridas a mayor rapidez (debido tanto a especialización como a concurrencia), menor costo de circuito y mayor economía de energía. El término *acelerador de hardware* se usa para cualquier recurso de hardware que ayude a un procesador de propósito general a ejecutar tipos específicos de operación más rápido. En este sentido, un procesador de señal digital que aumente el procesador simple de un teléfono celular se puede considerar como un acelerador de hardware. Otros aceleradores comunes se usan para las siguientes funciones:

- Renderizado (representación) de gráficos 3D.

- Compresión y descompresión de imágenes.

- Procesamiento de protocolo de red.

El creciente aumento en el número de transistores que se pueden poner en un chip hacen factible la integración de tales recursos de propósito especial o específicos de aplicación en el mismo chip que el procesador principal.

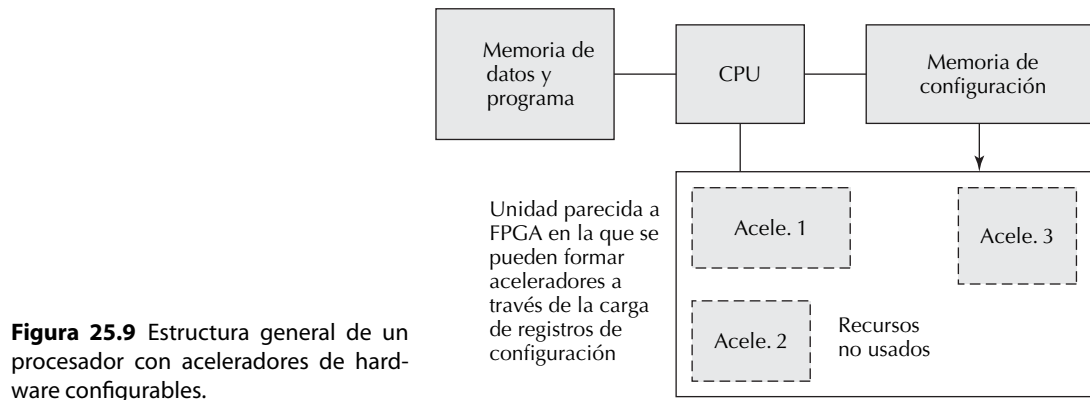


Figura 25.9 Estructura general de un procesador con aceleradores de hardware configurables.

Existen dos tendencias que permiten que tal integración de características específicas de aplicación sea económicamente viable. La primera tendencia es la comercialización de tales artículos como propiedad intelectual (diseño) en lugar de como productos terminados. El diseñador de un nuevo sistema toma varios diseños existentes, que acaso tengan interfaces comunes o estándar, y las combina en un chip. Por lo general, los diseños vienen con soporte de software o compilador, ello hace la integración mucho menos laboriosa. La segunda tendencia es la provisión de hardware configurable junto con el procesador que se puede convertir en unidades de propósito especial deseada a través de procedimientos de configuración especiales (figura 25.9). Entonces se puede elegir la configuración en la fábrica durante la fabricación, o se puede modificar dinámicamente al momento de correr bajo control de software. Como consecuencia de que el último enfoque requiere el establecimiento de registros de configuración, es un proceso más bien lento y no se puede realizar al nivel de instrucciones individuales; en vez de ello, con frecuencia se hace reconfiguración al momento de conmutar la tarea.

En el resto de esta sección se consideran los procesadores gráficos y de red como dos ejemplos específicos de aceleración de procesamiento mediante la proporción de recursos de hardware específicos de aplicación.

Un procesador gráfico tiene una arquitectura y las unidades de función requeridas que permiten que las tareas orientadas a gráficos se completen mucho más rápido que en un procesador de propósito general. Para el caso de tales tareas no es raro que un procesador gráfico tenga un pico de calificación en gigaflops que es mayor que el de un microprocesador de avanzada por un factor de diez o más. Por tanto, en virtud de que un procesador gráfico es comparable en complejidad de circuito y costo a un procesador de propósito general, ofrece una efectividad 10 veces mayor para aplicaciones con muchos gráficos. Esta efectividad de costo mejorada beneficia sistemas de muchos tipos, desde las máquinas de juegos de bajo costo hasta los sistemas de animación y visualización de alto perfil utilizados para estudios de cine. A diferencia de sus predecesores, que eran muy alambrados y arquitectónicamente cerrados, los procesadores gráficos más nuevos son programables y siguen estándares abiertos para conectividad. Por tanto, se han explorado muchos más usos para los modernos procesadores gráficos de alto rendimiento.

Las aplicaciones gráficas tratan con entidades geométricas como puntos y formas. Tanto la ubicación como los atributos gráficos de un punto (pixel) se pueden expresar como vectores numéricos. Las transformaciones (corrimiento, rotación, etc.) en los objetos gráficos se pueden formular como operaciones matriciales. Estas operaciones de datos intensos contienen muchas suboperaciones independientes que se pueden realizar en paralelo. Agregar textura a una imagen renderizada también requiere la recuperación de grandes volúmenes de datos de una memoria de textura especial y muchos cálculos

independientes. Actualmente, Nvidia (www.nvidia.com) y ATI (www.ati.com) ofrecen procesadores gráficos populares.

Un procesador de red monta entre una “línea” de red entrante y un procesador anfitrión (*host*). Tiene la responsabilidad de recepción y procesamiento en tiempo real de paquetes entrantes a la rapidez de línea. Si uno considera paquetes cortos de 64 B que llegan a la tasa de datos de 10 Gb/s, un tiempo de paquete es de aproximadamente 51 ns. Un procesador de 2 GHz que ejecuta un promedio de una instrucción por ciclo de reloj tiene tiempo para ejecutar aproximadamente 100 instrucciones para procesar tal paquete en tiempo real, para asegurar que los paquetes críticos no se dejen y que se pueden satisfacer varios requerimientos de calidad de servicio. No se puede lograr mucho con 100 instrucciones de un procesador RISC común; de ahí la necesidad para un procesador de red con un conjunto de instrucciones especializadas y otras capacidades. El reto al diseñar un procesador de red se encuentra en la necesidad simultánea de rendimiento extremadamente alto y flexibilidad para adaptarse a las necesidades de tasas de datos futuros y protocolos de red.

Con el propósito de lograr el mayor rendimiento necesario para continuar con la alta tasa de datos entrante, los procesadores de red están equipados con una o más de las características siguientes:

- Instrucciones para operaciones comunes relacionadas con comunicación como equiparación de bit, árbol de búsqueda y cálculos de verificación de redundancia cíclica.
- Paralelismo a nivel instrucción o multihilo para permitir el ocultamiento de acceso a memoria y otras latencias que conduzcan a rendimiento degradado.
- Procesamiento de paquete paralelo/encauzado, donde cada etapa de *pipeline* realiza una pequeña tarea y se usan múltiples *pipelines* para mejorar el rendimiento total.

Los procesadores de red los diseñan y comercializan Cisco Systems, Intel, IBM y otras compañías [Crow03]. La figura 25.10 muestra un diagrama de bloque simplificado del procesador de red de Cisco, Toaster2, que tiene la intención de ofrecer adelantamiento de paquete rápido con base en procesamiento de encabezado (*header*) de paquete. Como se ve en la figura 25.10, este procesador está compuesto de una matriz 4×4 de elementos de procesamiento simples que forman cuatro *pipelines* de flujo de datos paralelas. Los elementos de procesamiento (PE) en cada columna comparten un sistema de memoria jerárquica. Para cada paquete, se forma un contexto de 128 B que contiene el *header* del paquete y otra información de control. El contexto entra a una de las cuatro *pipelines* de procesamiento idéntico y se puede enviar desde el *buffer* de salida de vuelta al *buffer* de entrada si el trabajo de alguna etapa no

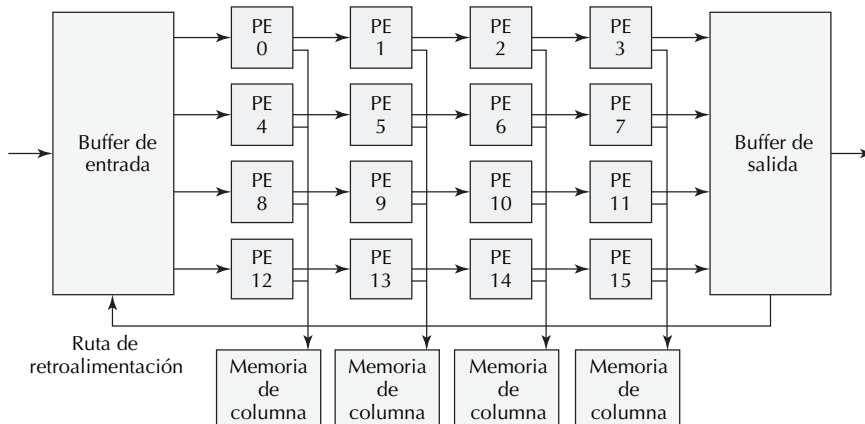


Figura 25.10 Diagrama de bloque simplificado de Toaster2, el procesador de red de Cisco.

se termina dentro del tiempo fijo que se le asignó. Cada elemento de procesamiento es un procesador VLIW con su propia memoria caché privada.

■ 25.6 Procesamientos vectorial, matricial y paralelo

En 1966, M. J. Flynn propuso una clasificación de cuatro vías de los sistemas de cómputo con base en las nociones de flujo de instrucciones y flujo de datos. La clasificación de Flynn se volvió estándar y se usa ampliamente. Flynn acuñó las abreviaturas SISD, SIMD, MISD y MIMD para las cuatro clases de computadoras que se muestran en la figura 25.11, con base en el número de flujos de instrucción (sencillo [*single*] o múltiple) y flujos de datos (sencillo [*single*] o múltiple). La clase SISD representa los sistemas “uniprosesor” ordinarios. Las computadoras en la clase SIMD, con muchos procesadores dirigidos por instrucciones emitidas desde una unidad de control central, a veces se caracterizan como “procesadores matriciales”. Las máquinas en la categoría MISD no han encontrado aplicación extensa, pero uno los puede ver como *pipelines* generalizadas en las que cada etapa realiza una operación relativamente compleja (en oposición a las *pipelines* ordinarias que se encuentran en los procesadores modernos, donde cada etapa efectúa una muy simple operación a nivel de subinstrucción).

La categoría MIMD aumentó en popularidad con los años de modo que abarca una amplia clase de computadoras. Por esta razón, en 1988 E. E. Johnson propuso una mayor clasificación de tales máquinas con base en su estructura de memoria (global o distribuida) y el mecanismo usado para comunicación/sincronización (variables compartidas [*shared variables*] o paso de mensaje [*message passing*]). De nuevo, una de las cuatro categorías (GMMP) no es muy utilizada. La clase GMSV es la que holgadamente se conoce como “multiprocesamiento (memoria compartida)”. En el otro extremo, la clase DMMP se conoce como “multicomputación (memoria distribuida)”. Finalmente, la clase DMSV, que se está volviendo popular en vista de la combinación de escalabilidad de memoria distribuida con la facilidad de programación del esquema de variable compartida, a veces se le llama “memoria compartida distribuida”. Cuando todos los procesadores en una máquina de tipo MIMD ejecutan el mismo programa, el resultado a veces se le refiere como “programa sencillo, datos múltiples” o SPMD.

En la categoría SIMD, el paralelismo resulta de la ejecución del mismo flujo de instrucciones en múltiples flujos de datos. Se discute la computación paralela SIMD y sus ventajas en el contexto de un ejemplo específico: multiplicar un vector coeficiente o ponderado por un vector de datos en una base

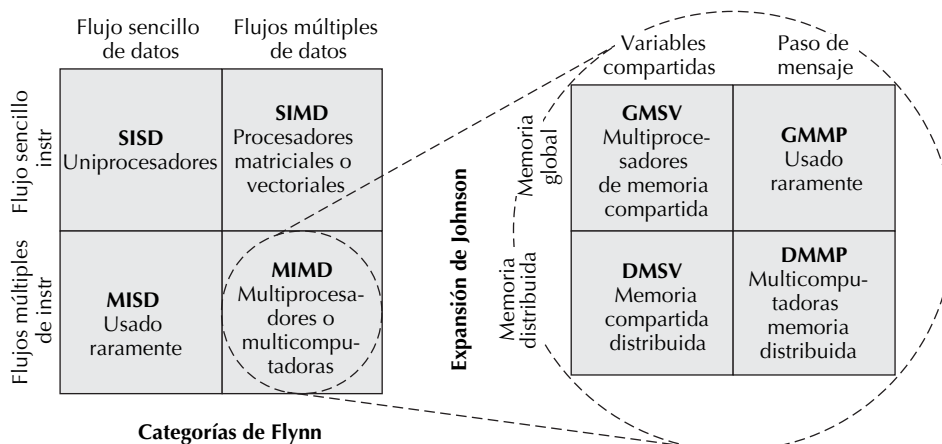


Figura 25.11 Clasificación Flynn-Johnson de sistemas de cómputo.

de elemento por elemento. En términos de hardware, la concurrencia puede ocurrir en el tiempo o en el espacio. La concurrencia en el tiempo conduce a *procesamiento vectorial*, donde un multiplicador sencillo profundamente encauzado se alimenta con los pares de números en turno, ello produce los productos en la salida en rápida sucesión. Con el fin de continuar con el alto rendimiento total del multiplicador, se ponen a disposición registros vectoriales especiales para retener los operandos y los resultados producidos. Adicionalmente, una memoria interpolada de alto rendimiento total se usa para suministrar a los registros los datos o para almacenar los resultados de vuelta en la memoria. El procesamiento vectorial difiere de la simple emisión de una larga secuencia de instrucciones *add* idénticas en una arquitectura escalar encauzada en las siguientes formas, que contribuyen al mayor rendimiento de un procesador vectorial:

Sólo una instrucción se lee (*fetch*) y decodifica para toda la operación.

Se sabe que las multiplicaciones son independientes (sin *checking* o *interlocks*).

El *pipelining* se aplica a los accesos a memoria, así como a la aritmética.

Los métodos adicionales de mejora del rendimiento incluyen el encadenamiento encauzado y la ejecución condicional, que se discutirán en la primera mitad del capítulo 26.

La concurrencia SIMD en el espacio conduce al *procesamiento matricial*. Al tomar el mismo ejemplo de multiplicación de elemento por elemento, con el párrafo anterior, los vectores coeficiente y de datos se pueden dividir entre p elementos de procesamiento, donde cada uno tiene su propia memoria y multiplicador. Si dichos vectores tienen longitud n , entonces cada uno de los p elementos de procesamiento será responsable de n/p multiplicaciones. La implementación y el rendimiento del procesador matricial se discutirán con más detalle en la segunda mitad del capítulo 26.

Las arquitecturas MIMD con una memoria global compartida se implementan al proporcionar una red de interconexión procesador a memoria que permita a todo procesador acceder a cualquier parte de una gran memoria física. La forma más simple de tal red es un bus común al cual se conectan cada procesador y módulo de memoria. Los *chips multiprocesadores* (múltiples CPU en un chip) usualmente siguen este enfoque. La memoria compartida ofrece un modelo de programación simple y limpio y permite que los procesadores pasen operandos y resultados unos a otros al escribir y leer de la memoria compartida. Para facilitar la congestión en el bus y reducir la latencia de acceso a memoria, que ya es un serio problema con un solo procesador, a cada uno de los procesadores se le proporciona una caché privada donde conserva sus valores usados con más frecuencia. Si los datos se deben compartir entre los procesadores para permitir su cooperación en la solución de problemas, en los diversos cachés pueden existir copias múltiples de los elementos de datos compartidos, lo anterior conduce al desafiante problema de la coherencia de caché. Los esquemas de interconexión punto a punto y las redes de conmutación multietapas ofrecen mayor ancho de banda agregado y facilitan el problema de congestión debido al uso de un bus común. Sin embargo, estos esquemas más elaborados también complican el reforzamiento de la coherencia de caché. El multiprocesamiento MIMD de memoria compartida se cubrirá en el capítulo 27.

En las arquitecturas MIDM con memoria distribuida, cada procesador y su unidad de memoria privada constituyen un nodo. Tales nodos, que pueden variar en número desde unos pocos hasta muchos miles, se comunican entre ellos a través de paso de mensaje explícito. Los procesadores se pueden interconectar mediante un bus simple, un esquema punto a punto o una red de conmutación multietapa. Las máquinas MIMD de memoria distribuida ofrecen excelente aceleración para aplicaciones que se pueden partir en grandes porciones, más o menos independientes, capaces de correr en diferentes nodos con mínima interacción. Las aplicaciones que son más intensas en comunicación pueden sufrir latencia excesiva al adquirir datos de *memorias remotas* (módulos de memoria en otros nodos, en

oposición a memoria local). Por tanto, para el éxito de este enfoque, son esenciales las estrategias para ocultar tales latencias de acceso remoto a través de traslapamiento con cálculos locales.

Una implementación actualmente popular de sistemas MIMD de memoria distribuida se basa en la interconexión de recursos de procesamiento y comunicación comerciales, cada uno con su aplicación local y software de sistema, bajo el control de un mecanismo de coordinación de nivel superior. El uso de PC o estaciones de trabajo comercialmente disponibles con conectividad Ethernet es un ejemplo. Los sistemas resultantes se conocen como *clusters* (grupos) o *redes de estaciones de trabajo* (COW, NOW). El enfoque de multicomputación distribuida para paralelismo MIMD es el tema del capítulo 28.

Aunque la figura 25.11 pone juntas todas las máquinas SIMD, de hecho hay variaciones similares a las sugeridas para computadoras paralelas MIMD. En otras palabras, puede haber máquinas SIMD de memoria compartida y memoria distribuida en las que los procesadores se comuniquen mediante variables compartidas o paso de mensaje explícito.

En términos de desarrollo de actividades y aplicaciones, el procesamiento paralelo SIMD y MIMD ha experimentado altas y bajas en lo que parece ser un ciclo de 20 años. Intenso interés en el procesamiento paralelo comenzó durante la década de 1960, con el desarrollo de la ILLIAC IV, una máquina de investigación construida en la Universidad de Illinois. Limitadas aplicaciones, dificultadas en desarrollo de software y alto costo de hardware enfriaron esta fase inicial de entusiasmo. Durante la década de 1980 existió mayor gasto para la defensa en Estados Unidos. La explosión resultante en fondeos de investigaciones, en combinación con la tecnología de compilador ampliamente mejorada, condujeron a interesantes aplicaciones de procesamiento paralelo y al surgimiento de numerosos desarrolladores y vendedores de computadoras paralelas. Los subsecuentes recortes de fondos, seguidos por quiebras de compañías, condujeron a renovado escepticismo acerca de la aplicabilidad de las técnicas de procesamiento paralelo más allá de ciertas áreas nicho. Durante la década de 2000, las necesidades de los portales de Internet y las grandes compañías de comercio electrónico condujeron a una renovada actividad en el procesamiento paralelo. Pero esta vez el interés se enfocó en el uso de procesadores comerciales, en lugar de en unidades diseñadas a la medida, junto con buses o conmutadores de ruta para construir un sistema distribuido a partir de componentes interconectados de costo bastante bajo.

Idealmente, debe ser posible configurar grandes sistemas paralelos a partir de unos más pequeños, tal como los niños construyen estructuras a partir de conectar bloques de juguete, con el rendimiento mejorado linealmente conforme se añade cada pieza. Sin embargo, debido a una combinación de inadecuado soporte de software y la maldición de la ley de Amdahl, este ideal de aceleración lineal todavía es una meta ilusoria. Recuerde de la sección 4.3 que si f representa la fracción del tiempo de ejecución de un programa debido a cálculos no paralelos, incluso suponiendo que el resto del programa goza de la aceleración perfecta de p cuando corre en p procesadores, la aceleración global sería:

$$s = \frac{1}{f + (1 - f)/p} \leq \min\left(p, \frac{1}{f}\right) \quad [\text{Fórmula de aceleración de Amdahl}]$$

Mientras que la fracción secuencial f de Amdahl parece condenar a la ruina cualquier intento de procesamiento paralelo para ciertas aplicaciones, es posible traslapar las partes secuenciales de un programa o tarea con partes enormemente paralelas de otros programas o tareas. Esto último conduce a procesamiento paralelo efectivo en costo; aun cuando la aceleración de cada aplicación pueda permanecer abajo de lo óptimo, el sistema global opera a alta eficiencia y rendimiento computacional total cerca del pico.

Para completar el gran cuadro, se discute brevemente la raramente usada organización MIMD en el contexto de un procesador de propósito especial. Una razón para la falta de popularidad del MISD consiste en que la mayoría de los problemas de aplicación no se mapean fácilmente en arquitectura MISD, ello hace difícil desarrollar una máquina MISD de propósito general. Sin embargo, en los sistemas de propósito especial, MISD puede ser viable. La figura 25.12 muestra un ejemplo de un flujo de datos sen-

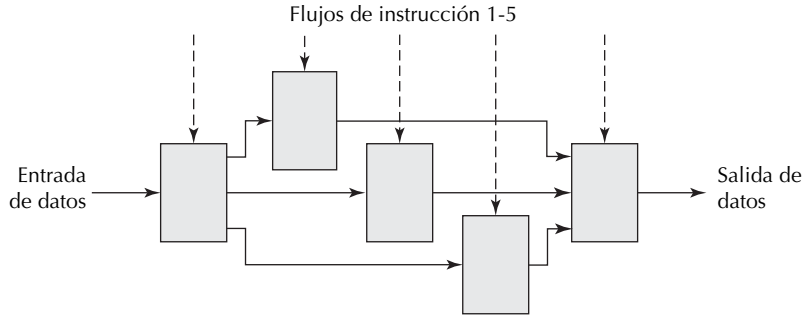


Figura 25.12 Múltiples flujos de instrucción que operan en un flujo de datos sencillo (MISD).

cillo que entra a una máquina MISD de cinco procesadores. El flujo de datos experimenta varias transformaciones y sale en el otro extremo como una secuencia de resultados. Las transformaciones pueden ser diferentes para cada elemento de datos, o debido a operaciones condicionales dependientes de datos en los flujos de instrucción (activadas por control) o con base en tags de control especiales llevadas por los datos (activadas por datos). Por tanto, la organización MISD se puede ver como una *pipeline* flexible con múltiples rutas de bifurcación y etapas programables. Una forma más simple de procesamiento MISD se encuentra en de cada hilera de PE en el procesador de red que se muestra en la figura 25.10.

PROBLEMAS

25.1 Consumo de energía en procesadores

Observe que la menor tasa de crecimiento para rendimiento por watt (diez cada diez años) frente al rendimiento absoluto (100 cada diez años) en la figura 25.1 sugiere que el consumo de energía absoluto de los procesadores de alto rendimiento ha crecido. Determine la tasa de crecimiento de disipación de energía y verifique esto al obtener el consumo de energía de los procesadores Pentium (versiones regulares no móviles) de la literatura.

25.2 Uso de instrucciones MMX

Use las instrucciones MMX de la tabla 25.1, así como instrucciones ordinarias como las definidas para MiniMIPS en la parte dos del libro, para realizar lo siguiente tan eficientemente como sea posible:

- Obtener del operando 8-vector 0000000a el resultado 8-vector aaaaaaaa.
- Obtener de los operandos 4-vector abcd y efgh el resultado 4-vector afch. *Sugerencia:* Use una máscara que tenga todos 1 y todos 0 en campos alternos de 16 bits [Pele97, p.31].
- Calcule la suma de los cuatro elementos en el 4-vector almacenado en un registro MMX.

- Calcule el producto interno de dos 4-vectores almacenados en registros MMX.
- Transponga una matriz 4×4 almacenada en los registros MMX del 0 al 3, donde cada hilera aparece en un registro [Pele97, p.33].

25.3 MiniMIPS 2D

Se va a implementar un MiniMIPS extendido que tenga instrucciones que operan sobre vectores 2D con elementos de tamaño mediapalabra almacenados en las mitades superior e inferior de registros de 32 bits.

- Presente el diseño de un sumador de 32 bits que realice sumas ordinarias de 32 bits, así como sumas vectoriales paralelas con resultados en ciclo, saturación sin signo y saturación con signo. *Sugerencia:* Piense en términos de un sumador con selección de acarreo.
- Discuta la implementación de una versión paralela de la instrucción `slt`.
- ¿Cómo se puede modificar el multiplicador hardware de la figura 11.4 para proporcionar capacidades de multiplicación paralela (baja y alta, como en MMX) además de la multiplicación ordinaria 32×32 ?

- d) Repita la parte c) para el multiplicador matricial de la figura 11.7.
- e) Discuta la implementación de una instrucción *multiply-add* paralela, que produzca un resultado de 32 bits.

25.4 Otras extensiones ISA para multimedia

Otros procesadores también tienen extensiones ISA para manipulación eficiente de aplicaciones multimedia. Estudie las siguientes extensiones y escriba para cada una un informe que recuerde la discusión de MMX en la sección 25.2. La referencia [Micr96] contiene artículos que describen las primeras tres de éstas.

- a) Conjunto de instrucción visual (VIS) de UltraSparc.
- b) Arquitectura MediaProcessor de MicroUnity.
- c) Extensión PA-RISC MAX-2 de Hewlett-Packard.
- d) Arquitectura AltiVec de Power PC.
- e) Extensión de flujo SIMD SSE-2 de Intel.
- f) Extensión de Intel para la arquitectura IA-64.

25.5 Emisión múltiple de instrucciones

Muestre cómo se puede derivar la curva de aceleración de la figura 24.5*b* a partir de la gráfica de barras de la figura 25.4*a*. *Sugerencia:* Para un ancho de emisión de instrucción de *i*, necesita imaginar la fracción de ciclos en los que *i* o más instrucciones están disponibles para ser emitidas.

25.6 Ejemplo de arquitectura VLIW

El Trimedia TM32 es un procesador VLIW que tiene calendarización completamente estática. Cada palabra de instrucción contiene cinco campos de operación, y el compilador llena las rendijas no utilizables con *no-op*. El compilador también asegura que las dependencias de datos se satisfagan a través de calendarización adecuada, dado que el procesador no tiene capacidad de detección de riesgo. Estudie el conjunto de instrucciones Trimedia TM32 con suficiente detalle para ser capaz de construir una secuencia de instrucciones para sumar dos vectores de longitud *n* con el uso de instrucciones similares a MiniMIPS en cada uno de los cinco campos de operación. Luego compare el resultado con el de una secuencia de instrucciones MiniMIPS para la misma suma vectorial, con y sin desenrollado de ciclo.

25.7 Arquitectura Itanium de Intel

La arquitectura Itanium de Intel permite que cinco operaciones se especifiquen en cada instrucción EPIC. Sin embargo, existen restricciones acerca de qué tipos de operación pueden aparecer en cada una de las cinco rendijas dentro de una instrucción.

- a) Estudie la arquitectura Itanium de Intel y presente en forma de tabla sus hallazgos acerca de los tipos de operación que se pueden especificar en cada rendija.
- b) Discuta por qué pudieron introducirse estas restricciones.
- c) Si cualquier rendija en una instrucción puede contener una operación aritmética, corrimiento o lógica de los tipos que se encuentran en el conjunto de instrucciones MiniMIPS de la tabla 6.2, discuta la relación del rendimiento de Itanium con la de MiniMIPS en aplicaciones de intenso cálculo que corren, cuando en sus implementaciones se usan tecnologías y circuitos comparables.
- d) Compare la filosofía de diseño básico de la arquitectura Itanium de Intel con la de la arquitectura AMD de siguiente generación [Webe01].

25.8 Emisión de múltiples instrucciones

Considere las siguientes dos secuencias de instrucciones y suponga que todas la instrucciones en cada secuencia caen en la ventana de emisión. Ignore todas las limitaciones de recurso a menos que se especifique explícitamente.

Secuencia 1	Secuencia 2
lw \$5, 0(\$4)	lw \$11, 0(\$10)
sub \$6, \$8, \$9	lw \$6, 4(\$9)
add \$10, \$7, \$7	add \$13, \$11, \$6
sw \$6, 4(\$15)	lw \$12, 8(\$15)
sw \$10, 0(\$14)	and \$14, \$16, \$17
add \$19, \$16, \$17	sub \$4, \$7, \$5
sub \$12, \$12, \$19	sw \$19, 0(\$4)
sw \$12, 8(\$13)	add \$25, \$11, \$6
	sw \$25, 4(\$26)
	sw \$27, 0(\$24)

- a) Para cada secuencia, determine el número mínimo de ciclos de emisión si las instrucciones se deben emitir en orden.
- b) Repita la parte a), pero suponga que en cada ciclo sólo se puede emitir una instrucción *load* o *store*.

- c) Repita la parte a), pero suponga emisión fuera de orden.
- d) Repita la parte c), pero suponga que en cada ciclo sólo se puede emitir una instrucción *load* o *store*.

25.9 Aliasing de registro

Debido al limitado número de registros en la mayoría de las máquinas, los programas tienden a reutilizar el mismo registro para retener varios valores temporales diferentes. Cuando se permite la ejecución de instrucciones fuera-de-orden, es muy posible que una o más de éstas que usen un valor temporal en $\$t1$ estén en varias etapas de ejecución cuando se emite una instrucción que escribe un valor temporal diferente en $\$t1$. A esta situación se le conoce como *aliasing de registro*. Discuta cómo esta situación puede crear problemas en la ejecución correcta de instrucción y cómo el renombrado de registro puede resolver el problema.

25.10 Hardware para predicción de valor

La predicción de último valor es útil cuando una instrucción produce repetidamente el mismo valor de resultado. La predicción de paso de valor (*stride-value*) es útil cuando una instrucción produce los valores $v, v + s, v + 2s, \dots$ en sucesión.

- a) Muestre cómo se puede implementar la predicción de último valor con una mecanismo parecido a caché.
- b) Discuta la implementación de la predicción de paso de valor. *Sugerencia:* Piense en términos de un diagrama de estado similar al usado en la predicción de bifurcación basada en historia.

25.11 Implementación de la tabla memo

La implementación de la tabla memo en la figura 25.8 supone un modo de operación asíncrono, donde el tiempo ahorrado por encontrar un valor de resultado en la tabla memo se traduce directamente en tiempo de ejecución reducido.

- a) ¿Cómo funcionaría este esquema en un sistema síncrono?
- b) Diseñe el circuito de control de la figura 25.8.
- c) ¿Una tabla memo sería útil para multiplicación o división en MiniMIPS? Discuta.

25.12 Rendimiento de la tabla memo

Una tabla memo funciona en forma muy parecida a una memoria caché. Por ejemplo, si se considera división, cuando el cociente requerido se presenta en la tabla memo, la operación toma un ciclo de reloj. De otro modo, el procesador realiza división base 2 o base 4 en, por decir, 20 ciclos de reloj. Esta “penalización por fallo” también es comparable con la que se causa al acceder a memoria principal cuando una palabra requerida no está en la memoria caché. Aunque hay una diferencia clave entre operación memo (*memoing*) y operación caché (*caching*): mientras que la memoria caché requiere una tasa de impacto bastante elevada (por decir, 80% o más) para ser efectiva, *memoing* puede ser muy beneficiosa incluso con una tasa de impacto de 50% o menos.

- a) Explique las razones para la diferencia entre *memoing* y *caching*.
- b) Cuantifique los beneficios de rendimiento de *memoing* en términos de la tasa de impacto h en la tabla memo.

25.13 Hardware suave

Se ha sugerido que se pueden usar dispositivos parecidos a FPGA para resolver el problema de obsolescencia de hardware; el hardware se adquiere en la forma de una colección amorfa de celdas de computación que se configuran en un procesador particular mediante la carga de sus registros de configuración con un patrón específico de 0 y 1. Entonces el hardware se puede actualizar en forma muy parecida al software, a saber, comprando o descargando gratuitamente extensiones y parches desde el sitio web del fabricante. Argumente que, aunque este enfoque puede ser útil para corregir *bugs* y agregar ciertas capacidades no previstas al momento de diseño, no resuelve completamente el problema de obsolescencia.

25.14 Procesadores gráficos

Responda las siguientes preguntas para un procesador gráfico de su elección.

- a) ¿Qué tipos de operandos de punto fijo o punto flotante soporta?
- b) ¿Qué recursos de hardware, más allá de lo que está disponible en un procesador de propósito general, se usan para acelerar cálculos numéricos?

- c) ¿Qué instrucciones y recursos específicos se proporcionan para facilitar escalamiento y rotación de imagen?
- d) ¿Cómo se facilitan el sombreado, la remoción de línea oculta y la presentación de superficies texturizadas?
- e) ¿Cómo interactúa el procesador gráfico con el resto del sistema de cómputo?

25.15 Procesadores de red

- a) El procesador de red Toaster2 de Cisco Systems se describió brevemente en la sección 25.5. Estudie este procesador de red y describa, en dos o tres páginas, la estructura interna de los PE y sus interacciones con las memorias de columna y los *buffer* de entrada/salida.
- b) Para otro procesador de red de su elección, describa la arquitectura del hardware y contraste su operación con la del Toaster2 de Cisco.
- c) Mientras que los procesadores gráficos aparecieron primero como unidades de hardware inflexibles y dedicadas y sólo recientemente se desarrollaron como dispositivos programables, los procesadores de red fueron programables desde el principio. ¿Puede explicar la razón para esta diferencia?

25.16 SIMD frente a MIMD

La comunidad de arquitectura de computadoras ha debatido durante mucho tiempo los méritos relativos de

las computadoras paralelas SIMD frente a MIMD. Las primeras computadoras paralelas eran predominantemente máquinas SIMD, mientras que los sistemas modernos casi son exclusivamente MIMD. Sólo la implementación de procesador vectorial del concepto SIMD todavía está en amplio uso en las máquinas de propósito general. Estudie este debate y prepare un reporte de cinco páginas sobre él, dedicando aproximadamente una página a cada una de las siguientes preguntas:

- a) ¿Por qué las arquitecturas SIMD perdieron apoyo para las máquinas de propósito general?
- b) ¿En qué dominios de aplicación todavía se usan arquitecturas SIMD?
- c) ¿Por qué se usan las arquitecturas SIMD, y es probable que se vuelven todavía más populares, en aceleradores de hardware?

25.17 Fórmula de aceleración de Amdahl

- a) Demuestre que la fracción secuencial f de la fórmula de aceleración de Amdahl es aproximadamente $1/s - 1/p$ cuando p es grande.
- b) ¿Qué valor produce la fórmula de la parte a) para f cuando la aceleración es $s = 80$ con procesadores $p = 100$? ¿Cuál es el error en este caso?
- c) Derive una expresión de error cuando $1/s - 1/p$ se usa para aproximar f . Escriba su expresión para hacer bastante obvio que el error relativo es pequeño para p grande.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Bour03] Bourianoff, G., "The Future of Nanocomputing", *IEEE Computer*, vol. 36, núm. 8, pp. 44-53, agosto de 2003.
- [Cho01] Cho, S., P.-C. Yew y G. Lee, "A High-Bandwidth Memory Pipeline for Wide Issue Processors", *IEEE Trans. Computers*, vol. 50, núm. 7, pp. 709-723, julio de 2001.
- [Crow03] Crowley, P., M. A. Franklin, H. Hadimioglu y P. Z. Onufryk, *Network Processor Design: Issues and Practices*, vol. 1, Morgan Kaufmann, 2003.
- [Flynn96] Flynn, M. J. y K. W. Rudd, "Parallel Architectures", *ACM Computing Surveys*, vol. 28, núm. 1, pp. 67-70, marzo de 1996.
- [Fran03] Franklin, M., *Multiscalar Processors*, Kluwer, 2003.
- [John91] Johnson, M., *Superscalar Microprocessor Design*, Prentice Hall, 1991.
- [Kuma03] Kumar, R., K. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Processor Power Reduction via Single-ISA Heterogeneous Multi-Core Architectures", *Computer Architecture Letters*, vol. 2, núm. 1, pp. 2-5, julio de 2003.

- [Lee02] Lee, R., "Media Signal Processing", *Section 39.1 in The Computer Engineering Handbook*, V. G. Oklobdzija, ed., pp. 39-1 a 39-38, CRC Press, 2002.
- [Lo97] Lo, J. L., S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen, "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading", *ACM Trans. Computer Systems*, vol. 15, núm. 3, pp. 322-354, agosto de 1997.
- [Marr02] Marr, D. T., *et al.*, "Hyper-Threading Technology Architecture and Microarchitecture", *Intel Technology J.*, vol. 6, núm. 1, 14 de febrero de 2002. Disponible en: <http://www.intel.com/technology/itj/>
- [Micr96] IEEE Micro (Número especial acerca de Procesamiento Multimedia), vol. 16, núm. 4, agosto de 1996.
- [Mudg01] Mudge, T., "Power: A First-Class Architectural Design Constraint", *IEEE Computer*, vol. 34, núm. 4, pp. 52-58, abril de 2001.
- [Pele97] Peleg, A., S. Wilkie y U. Weiser, "Intel MMX for Multimedia PCs", *Communications of the ACM*, vol. 40, núm. 1, pp. 25-38, julio de 1997.
- [Rone01] Ronen, R., A. Mendelson, K. Lai, S.-L. Lu, F. Pollack y J. P. Shen, "Coming Challenges in Microarchitecture and Architecture", *Proceedings of the IEEE*, vol. 89, núm. 3, pp. 325-340, marzo de 2001.
- [Thak99] Thakkar, S. T y T. Huff, "Internet Streaming SIMD Extensions", *IEEE Computer*, vol. 32, núm. 12, pp. 26-34, diciembre de 1999.
- [Unge02] Ungerer, T., B. Robic y J. Silc, "Multithreaded Processors", *Computer J.*, vol. 45, núm. 3, pp. 320-348, 2002.
- [Webe01] Weber, F., "AMD's Next Generation Microprocessor Architecture", available from: http://www.x86-64.org/documentation_folder/
- [WWW] Información acerca de actuales computadoras de alto rendimiento está disponible en: www.top500.org.
- [Zyub01] Zyuban, V. V. y P. M. Kogge, "Inherently Lower-Power High-Performance Superscalar Architectures", *IEEE Trans. Computers*, vol. 50, núm. 3, pp. 268-285, marzo de 2001.

PROCESAMIENTO VECTORIAL Y MATRICIAL

“Cualquier tarea matemática podría, en principio, resolverse mediante conteo directo. Sin embargo, existen algunos problemas de conteo que dentro de poco podrán resolverse en pocos minutos, pero para los cuales, sin un método matemático, una vida no sería suficiente.”

Ernst Mach, 1896

“Imagina un gran salón como un teatro... Las paredes de esta cámara están pintadas para formar un mapa del globo... Una miríada de computadoras funcionan en el clima de la parte del mapa donde cada una está asentada, pero cada computadora sólo atiende a una ecuación o parte de ésta.”

L. F. Richardson, meteorólogo británico, fantaseando acerca de lo que más tarde se conocería como SIMD o procesamiento matricial

TEMAS DEL CAPÍTULO

- 26.1** Operaciones sobre vectores
- 26.2** Implementación de procesador vectorial
- 26.3** Rendimiento del procesador vectorial
- 26.4** Sistemas de control compartido
- 26.5** Implementación de procesador matricial
- 26.6** Rendimiento de procesador matricial

Una forma de mejorar el rendimiento de computadoras consiste en dejar que un solo flujo de instrucciones opere en múltiples flujos de datos. Este modo SIMD de computación se puede orquestar en dos formas. Primero, se pueden explotar operaciones repetitivas sobre elementos vectoriales para llenar continuamente una *pipeline* y sin peligro de riesgos de datos o control. Este método, junto con una memoria de gran ancho de banda y una provisión de encadenamiento encauzado (realizando múltiples operaciones sin que se almacenen valores intermedios en registros) permite a las supercomputadoras vectoriales lograr su alto rendimiento. Segundo, se pueden controlar múltiples ALU independientes mediante una sola unidad de control que transmita señales de control y efectúa la conclusión de muchas operaciones en la misma cantidad de tiempo en que una ALU completaría una sola operación. Ésta es la estrategia seguida en los procesadores matriciales.

26.1 Operaciones sobre vectores

Como se discutió en la sección 25.6, las operaciones encauzadas sobre vectores pueden conducir a un rendimiento significativamente mayor que el alcanzable en *pipelining* ordinario. En los lenguajes de programación de alto nivel ordinarios, las operaciones vectoriales se definen mediante ciclos e índices de ciclos. Por ejemplo, multiplicar un vector coeficiente o ponderado por un vector de datos sobre una base elemento por elemento se puede expresar mediante el ciclo en lenguaje de alto nivel:

```
for i = 0 to 63 do
    P[i] := W[i] × D[i]
endfor
```

Esta operación, expresada en forma vectorial como $P := W \times D$, se traduce a un ciclo en lenguaje ensamblador que realiza dos cargas, una multiplicación, un incremento del índice y una instrucción *branch* condicional que verifica para determinar si se han realizado suficientes iteraciones para salir del ciclo. Dentro de cada iteración hay paralelismo en el nivel de instrucción muy limitado. Incluso con desenrollado de ciclo, gran cantidad de instrucciones todavía se deben leer (*fetch*), decodificar y ejecutar individualmente.

En un procesador vectorial, la operación anterior consistiría de dos cargas de registro vectoriales (dando registros vectoriales de 64 elementos), una multiplicación vectorial y un almacenamiento vectorial. Si supone dos canales de acceso a memoria, cuya implementación se discutirá en la sección 26.2, la operación consistirá de tres fases (carga, multiplicación y almacenamiento):

```
load W
load D
P := W × D
store P
```

Cada una de estas cuatro operaciones vectoriales está encauzada, de modo que en cada ciclo de reloj se manipula un nuevo valor u operación. Desde luego, existen latencias de arranque y drenado de *pipeline* al comienzo y al final de cada operación vectorial, pero se les ignorará por ahora. Más allá de la ventaja del *pipelining* directo del procesamiento vectorial, se pueden usar algunos otros métodos para mejorar aún más el rendimiento. Por ejemplo, las fases de carga y almacenamiento se pueden traslapar con la ejecución de otras operaciones vectoriales, siempre que haya suficiente suministro de registros vectoriales. De igual modo, en lugar de esperar que se inicie la *pipeline* de multiplicación hasta que los vectores C y D se hayan cargado completamente, se puede iniciar después de que unos cuantos elementos de cada uno hayan quedado disponibles en los registros. El último método, conocido como *encadenamiento de pipeline*, obviamente se puede extender a la *pipeline* de almacenamiento, de modo que el almacenamiento resultante puede iniciar mucho antes de la multiplicación o de que se hayan completado las cargas a registro.

Las operaciones vectoriales o se escriben directamente en los programas, con el uso de constructos de programación paralela, o se deducen mediante compiladores de ciclos dentro de programas ordinarios. En el último caso, la reformulación de una computación secuencial como una computación vectorial no siempre es tan directa como para $P := W \times D$. Considere, por ejemplo, el siguiente ciclo en el que hay dependencia de datos de una iteración a la siguiente:

```
for i = 0 to 63 do
    A[i] := A[i] + B[i]
    B[i+1] := C[i] + D[i]
endfor
```

Por el momento, suponga distintos vectores de 64 elementos sin traslapamiento en memoria; además, no se preocupe por exceder las fronteras de A y B en la iteración final del ciclo debida a la expresión del índice $i+1$. Observe que $B[i+1]$, calculado en el segundo enunciado del ciclo, se usa al inicio de la siguiente iteración cuando el índice de ciclo avanzó a $i+1$. Tal dependencia *portada por ciclo* no implica que el ciclo no se puede paralizar. En este caso, el ciclo se puede paralizar, porque una vez que $A[i]$ se actualiza en una iteración con base en el valor de $B[i+1]$ de la iteración anterior, el nuevo valor resultante de $A[i]$ no se usa de nuevo. Proporcionar los detalles de esta paralelización se deja como ejercicio.

Considere, en contraste, el siguiente ciclo en el que hay dependencias de datos tanto dentro de una iteración como de una iteración a la siguiente:

```
for i = 0 to 63 do
  X[i+1] := X[i] + Z[i]
  Y[i+1] := X[i+1] + Y[i]
endfor
```

Observe que $X[i+1]$, calculado en el primer enunciado del ciclo, se usa más tarde en la misma iteración, así como al principio de la siguiente cuando el índice de ciclo avanzó a $i+1$. En este caso, debido a las dependencias circulares portadas por ciclo para X y Y, el ciclo no se puede paralelar.

Con frecuencia tales cálculos iterativos no vectoriales se puedan escribir en formas alternas que permitan procesamiento vectorial eficiente. Sin embargo, encontrar estas formas no siempre es sencillo.

Además de las operaciones vectoriales con dos operandos similares y un resultado vectorial, en la práctica se han encontrado útiles otras operaciones semejantes. Éstas incluyen:

Vector combinado con un valor escalar, que produce un resultado vectorial ($A := c \times B$).

Operación vectorial de un operando con un resultado vectorial ($A := \sqrt{B}$).

Operación reducción sobre un vector, que produce un resultado escalar ($s := \sum A$).

Por ejemplo, el producto interno de dos vectores X y Y se puede sintetizar a partir de una operación vectorial $Z := X \times Y$ seguida por una operación de reducción de suma $s := \sum Z$.

Los cálculos en matrices 2D y de más dimensiones se pueden expresar en términos de operaciones vectoriales sobre subarreglos (por ejemplo, hileras y columnas para el caso de matrices). Un vector que represente un subarreglo usualmente se especifica mediante una *dirección base* (punto de partida) y un *paso* (separación de dos elementos vectoriales consecutivos en memoria). Por tanto, un procesador vectorial puede manipular no sólo operaciones vectoriales, sino también operaciones sobre matrices y otros arreglos.

Ejemplo 26.1: Acceso a subarreglos matriz Considere una matriz A de $n \times n$ almacenada en memoria en orden de hilera mayor, ello significa que todos los elementos de la hilera i aparecen en su orden natural antes de los de la hilera $i + 1$. La memoria es direccionable por palabra, y cada elemento de matriz ocupa una palabra. Por tanto, el paso para acceder a los elementos de una hilera dada es 1. Si el elemento $A[0,0]$ se ubica en la dirección base b :

- Determine la dirección para el elemento $A[i,j]$.
- Encuentre el paso para acceder a los elementos de la columna j .

- c) Determine si, con un paso fijo, se puede acceder a los elementos de la diagonal principal de la matriz.
- d) Haga lo mismo para la antidiagonal principal.

Solución: Los elementos de A están almacenados en las n^2 localidades de memoria consecutivas, desde b hasta $b + n^2 - 1$.

- a) Hay i hileras, con ni elementos, antes de la hilera i . Si se toman en cuenta los j elementos en la hilera i que precede al elemento $A[i, j]$, la dirección del último elemento es $b + ni + j$.
- b) Los elementos de la columna j se almacenan en las direcciones $b + j, b + n + j, b + 2n + j, \dots$, lo que da un paso de n .
- c) A los elementos $A[i, i]$ de la diagonal principal, ubicados en las direcciones $b, b + n + 1, b + 2n + 2, \dots$, se puede acceder con un paso de $n + 1$.
- d) A los elementos $A[i, n - 1 - i]$ de la antidiagonal se puede acceder con un paso de $n - 1$.

Con base en la discusión anterior, el término “vector” se puede usar para referirse a un vector ordinario, almacenado en memoria con un paso de 1, así como a un subarreglo que tenga un paso arbitrario s . En la generación de direcciones de memoria para acceder a elementos de un vector con dirección base b y paso s , un registro de dirección dentro del procesador se inicializa a b y aumenta por s después de cada acceso. Al usar estas direcciones, los elementos del vector se leen (*fetch*) en lugares consecutivos de un registro vectorial en su totalidad, la longitud de la operación vectorial se controla al inicializar un registro de longitud de vector, que disminuye en 1 después de que cada elemento vectorial se envía a la unidad de función. Los vectores más largos se procesan en piezas que encajan en registros vectoriales. Por ejemplo, un vector con longitud 100 se puede cargar en dos registros vectoriales de longitud 64, con la longitud del vector especificada como 64 y 36 en el procesamiento de los dos subvectores, respectivamente. Este tipo de partición la realiza el compilador; el programador en lenguaje de alto nivel no necesita preocuparse por los detalles, pero debe estar atento a la penalización de rendimiento al usar un vector de 65 elementos frente a uno de longitud 64.

26.2 Implementación de procesador vectorial

Con base en la discusión de la sección 26.1, existen tres características clave de un procesador vectorial:

Unidades de función profundamente encauzadas para alto rendimiento total (ciclo de reloj muy corto).

Abundancia de registros vectoriales largos para alimentar las unidades de función y recibir los resultados.

Rutas de transferencia de datos de gran ancho de banda entre memoria y registros vectoriales.

El *encadenamiento encauzado*, o adelantar los valores de resultado calculados de una *pipeline* de unidad de función a otra, aunque no absolutamente necesario, es una característica muy útil. Cualquier procesador vectorial también debe tener una unidad escalar de alto rendimiento, porque de otro modo una pequeña fracción de las operaciones escalares mezcladas con las operaciones vectoriales reduciría severamente el rendimiento total (ley de Amdahl).

Ejemplo 26.2: Rendimiento de procesador vectorial Una aplicación específica corre en un microprocesador escalar que emplea 20% de su tiempo de ejecución en operaciones de verificación (*house-keeping*) y operaciones escalares, y 80% en operaciones vectoriales. Un procesador vectorial ofrece una

aceleración promedio de 4 para las operaciones vectoriales, pero frena las partes no vectoriales por un factor de 2 debido a su unidad escalar más simple. ¿Cuál es la aceleración esperada del procesador vectorial sobre el procesador superescalar? ¿Cuál mezcla de tiempos de operación escalar y vectorial no conduciría a mejora en rendimiento en el procesador vectorial?

Solución: El tiempo de ejecución superescalar de $0.2 + 0.8$ se convierte en $0.2 \times 2 + 0.8/4 = 0.6$ en el procesador vectorial. Por tanto, la aceleración es $1.0/0.6 = 1.67$. No habrá aceleración si la fracción x de tiempo de ejecución debido a operaciones no vectoriales satisface $2x + (1 - x)/4 = 1$, lo que conduce a $x = 0.43$.

En el estudio del *pipelining*, se trata con parámetros de diseño y negociaciones que influyen en la elección del número de etapas de *pipeline* en el contexto de implementación de procesador escalar o superescalar (sección 15.6). En las unidades de función encauzadas de procesadores vectoriales, las negociaciones son diferentes. El atasco no es un problema porque las operaciones vectorizadas son por definición libres de dependencias de datos. Sin embargo, las cabeceras de arranque y drenado de *pipeline* favorecerían *pipelines* más cortas, a menos que los vectores en promedio fuesen bastante largos. Este es el caso en muchos cálculos científicos y de ingeniería, que forman los dominios de aplicación principales para los procesadores vectoriales. Tener de media docena a 20 etapas en cada unidad de función no es raro en absoluto. La suma y la multiplicación usualmente tienen *pipelines* con menor número de etapas, mientras que la división y carga/almacenar tienden a tener *pipelines* más largas. Como consecuencia de que el ancho de banda de memoria representa el factor limitante en un procesador encauzado, las *pipelines* más profundas pueden ser útiles sólo si los registros vectoriales se pueden llenar y vaciar a una tasa alta.

La figura 26.1 muestra algunos de los componentes clave de un procesador vectorial simplificado. Existen tres unidades de función, que pueden realizar varias operaciones aritméticas. Las unidades de función se alimentan a partir de un archivo de registro vectorial. Parte del proceso de establecimiento de *pipeline* consiste en especificar el registro vectorial a usar por cada operando, las entradas iniciales en los registros y la longitud del vector. Luego, en cada ciclo de reloj, los contenidos de los registros vectoriales indicados se envían a una de las unidades de función y se actualizan los números de entrada de registro. Este proceso continúa hasta que se ha procesado el número de elementos especificado. La figura 26.1 también muestra tres multiplexores que permiten que las salidas de una *pipeline* se adelante

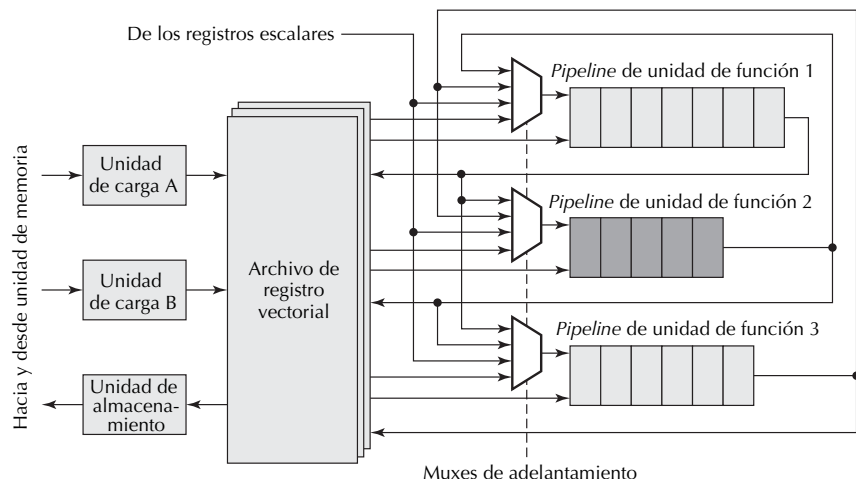


Figura 26.1 Estructura genérica simplificada de un procesador vectorial.

a cualquiera de las otras dos *pipelines* para encaadenamiento encauzado. También es posible suministrar un valor escalar como uno de los dos operandos en un cálculo vectorial. De nuevo la cadena encauzada se establece en el inicio. Por ejemplo, si la salida de la unidad de función 1 se va a enviar a la unidad de función 2, el segundo operando del último, que viene de un registro vectorial, se debe retardar de modo que el primer resultado de la unidad de función 1 y la primera entrada del registro vectorial especificado lleguen en el mismo ciclo de reloj. Esto es sencillo de verificar, dadas las profundidades de cada una de las *pipelines* y sus cabeceras de arranque.

El archivo de registro vectorial se puede ver como un arreglo 2D de registros. Si hay ocho registros vectoriales de longitud 64 en el archivo de registro, entonces cada registro está especificado por un campo de tres bits en instrucciones de máquina vectorial, mientras que cada elemento de un registro requiere un índice de seis bits para acceder dentro de un ciclo de reloj particular en el curso de ejecución de la instrucción vectorial. Usualmente, al menos tres tipos diferentes de operación vectorial suceden simultáneamente en un procesador vectorial (para ocultar la latencia de acceso a memoria):

Una operación de unidad de función sobre operandos vectoriales, que regresa un resultado vectorial.

Carga de registros vectoriales en preparación para futuras operaciones vectoriales.

Almacenamiento de resultados de una operación previa desde un registro vectorial hacia memoria.

Por tanto, un procesador vectorial contiene un mínimo de ocho registros vectoriales. Desde luego, más registros vectoriales pueden ser útiles para mantener ocupado mayor número de unidades de función, pero ello también complica la lógica de enrutamiento (*routing*) entre el archivo de registro y las unidades de función.

Para conocer por qué, considere que en la figura 26.1 cualquier registro vectorial será capaz de servir como el operando superior o inferior para cada una de las tres unidades de función o para recibir sus resultados. Al contar las unidades de carga y almacenamiento mostradas, el archivo de registro debe tener siete puertos de lectura y cinco de escritura. Si cada registro vectorial se ve como un archivo de registro escalar de tamaño 64, implementado como en la figura 2.9a, entonces, con ocho registros vectoriales, se necesita un conmutador de barra cruzada de 16×7 para enrutar las 16 posibles salidas de registro hacia los siete puertos de salida. De igual modo, en el lado de entrada del archivo de registro se necesita un conmutador de barra cruzada de 8×5 .

Proporcionar ancho de banda de memoria adecuado para continuar con las altas tasas de procesamiento de muchas unidades de función profundamente encauzadas es quizá la parte más desafiante del diseño de un procesador vectorial. Usualmente, para operandos vectoriales no se usa memoria caché de datos y el procesador accede directamente a la memoria principal. Para comenzar, la memoria está enormemente interpolada, 64-256 bancos de memoria no son raros en absoluto. Recuerde de la discusión de interpolación en la sección 17.4, que una memoria interpolada de 64 vías puede proporcionar datos a una tasa pico que sea 64 veces la de un módulo de memoria simple. Mantener esta tasa pico requiere que a un banco de memoria no se acceda de nuevo antes de que los otros bancos hayan tenido accesos. Este requisito se satisface claramente en accesos secuenciales o de paso 1. De igual modo, cualquier paso que sea relativamente primo con respecto al ancho de interpolado conduce al máximo ancho de banda de memoria. Infortunadamente, esta condición no ocurre de manera natural en las aplicaciones, pues el ancho de interpolado usualmente es una potencia de 2 y los pasos usados más comúnmente también son potencias de 2 o, al menos, pares.

Se ha diseñado un amplio arreglo de estrategias de ordenamiento de datos para lidiar con problemas de ancho de banda de memoria. Por ejemplo, si se almacena una matriz de 64×64 en orden de hilera mayor (figura 26.2a), el paso para acceder a las columnas será 64, que es el peor valor posible para una memoria interpolada de 64 vías: todos los accesos para leer una columna de la matriz se dirigirán al mismo banco de memoria, que obviamente no puede proporcionar un nuevo elemento en cada ciclo

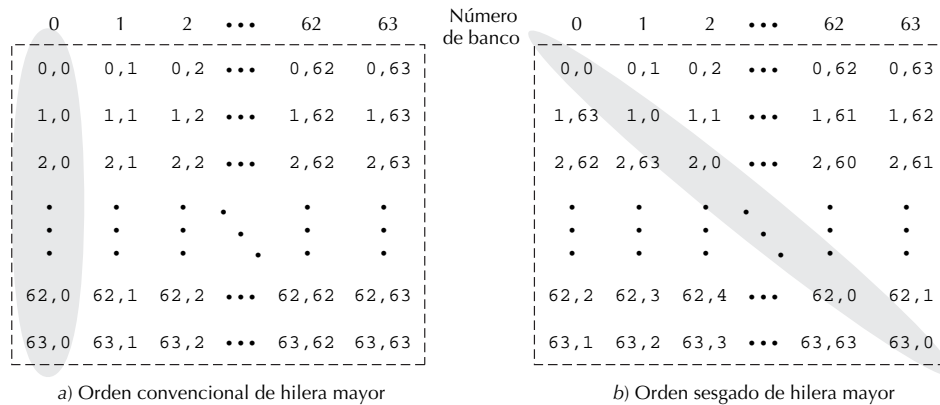


Figura 26.2 Almacenamiento sesgado de los elementos de una matriz de 64×64 para acceso a memoria libre de conflicto en una memoria interpolada de 64 vías. En ambos diagramas se resaltan los elementos de la columna 0.

de reloj. Además, si se usa el esquema de almacenamiento sesgado de la figura 26.2b, donde los elementos de la hilera i están rotados hacia la derecha por i posiciones, entonces el paso para el acceso a columna se vuelve 65. Esto último conduce a acceder a diferentes bancos de memoria al leer los 64 elementos de una columna. Desde luego, se agregará cierta cabecera porque es necesaria una traducción más compleja entre los índices hilera/columna y las direcciones de memoria, pero la cabecera es pequeña en relación con las ganancias.

Las instrucciones de máquina vectorial son similares a las de Intel MMX citadas en la tabla 25.1. Una diferencia clave es que los vectores en MMX son más bien cortos y fijos en longitud, mientras que, en una máquina vectorial, usualmente son largos y de longitud variable. Por tanto, la longitud vectorial con frecuencia se especifica como un parámetro o explícitamente dentro de la instrucción, o implícitamente a través de definiciones de datos iniciales. Las instrucciones típicas, distintas a las aritméticas sobre operandos vectoriales, o un vector y un operando escalar, incluyen las siguientes:

Carga y almacenamiento vectorial, con paso implícita o explícitamente inespecífico.

Carga y almacenamiento vectorial, con direcciones de memoria especificadas en un vector índice.

Comparaciones vectoriales (muchos tipos) sobre una base de elemento por elemento.

Copiado de datos entre registros, incluidos registros de longitud de vector y de máscara.

El registro de máscara de vector permite la realización selectiva de operaciones sobre elementos vectoriales, con base en el bit asociado en un registro de máscara. Con esta capacidad de enmascaramiento, la ejecución de una operación sobre elementos vectoriales especificados se puede predicar sobre una condición dependiente de datos previamente evaluada (por ejemplo, para evitar división entre cero).

Esta sección concluye con el trazado de pasos en la ejecución de una instrucción vectorial como $Z := X + Y$, donde la longitud del vector no necesariamente es un múltiplo entero de la longitud de registro vectorial, que se supone es 64. El hardware segmenta los vectores en subvectores de 64 elementos, donde los primeros elementos quizá contienen menos de 64 elementos. Entonces carga los registros vectoriales con segmentos de los dos vectores X y Y , a la vez, comenzando con los segmentos iniciales, tal vez irregulares. La misma segmentación se aplica al vector resultado Z , que se almacena en memoria conforme cada uno de sus segmentos se calcula en un registro vectorial. La figura 26.3 muestra cómo las operaciones de carga y almacenamiento de registro se traslapan con la suma al usar los registros vectoriales en un esquema de *doble buffering*.

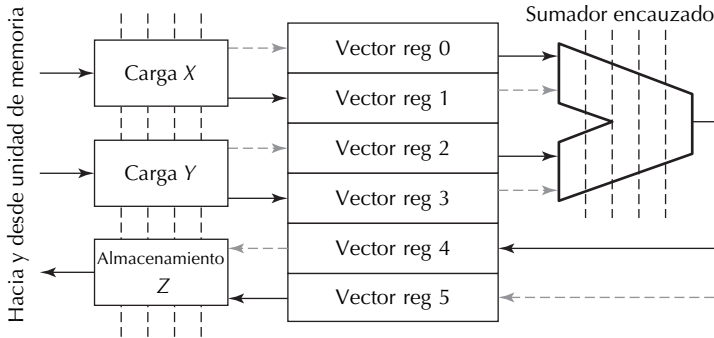


Figura 26.3 Procesamiento vectorial a través de carga/almacenamiento segmentado de vectores en registros en un esquema de doble buffering. Las líneas sólidas (punteadas) muestran flujo de datos en el segmento actual (siguiente).

26.3 Rendimiento del procesador vectorial

Como con la mayoría de las supercomputadoras, el rendimiento de los procesadores vectoriales con frecuencia se especifica en términos de FLOPS pico, que se obtienen cuando se ejecuta un programa especialmente hecho a la medida que ejercita de manera óptima todos los cálculos y evita problemas. Las cifras de rendimiento más realistas en *benchmarks* o aplicaciones reales puede ser más bajo que el rendimiento pico publicitado por uno o más órdenes de magnitud. Si la frecuencia de reloj de un procesador vectorial es x GHz, entonces cada una de sus unidades de función puede ejecutar x GFLOPS, siempre que se alimente continuamente con datos (vectores infinitamente largos). El rendimiento real es menor por una diversidad de razones.

Primero, cada operación vectorial tiene un tiempo de arranque (*start-up*) que se emplea incluso para vectores muy cortos. El efecto del tiempo de arranque sobre el rendimiento se vuelve despreciable sólo cuando los vectores son extremadamente largos. De hecho, para vectores muy cortos, el mejor rendimiento con frecuencia se obtiene con el uso de la unidad escalar de un procesador vectorial en lugar de la unidad vectorial. El tiempo de arranque incorpora la latencia de *pipeline* antes de que se produzca el primer resultado, así como el tiempo de establecimiento para longitud del vector y otros registros de control. Parte o todo el tiempo de establecimiento se puede traslapar con la ejecución de otras operaciones vectoriales, pero la cabecera de latencia de *pipeline* es inevitable. El tiempo de arranque para cada tipo de operación con frecuencia se especifica en términos de ciclos de reloj y rangos de unos cuantos a muchas decenas de ciclos.

Una forma útil para caracterizar el impacto de la cabecera de arranque sobre el rendimiento es a través de la noción de *longitud de vector de medio rendimiento*, que se define como la longitud de un vector que hace que el rendimiento sea la mitad de la lograda con vectores infinitamente largos. Las longitudes de vector de medio rendimiento varían desde decenas hasta alrededor de 100 en máquinas vectoriales reales, donde los valores más largos corresponden a mayor cabecera de arranque.

El segundo factor que limita el rendimiento de un procesador vectorial en comparación con su potencial pico es el tiempo de inactividad de recurso y la contención. Como indica el diagrama de tiempos de la figura 26.4, las *pipelines* de unidad de función multiplicar y sumar pueden estar ocupadas al mismo tiempo para la versión encadenada del cálculo $S := X \times Y + Z$. Sin embargo, si el cálculo de interés es $S := V \times W + X \times Y$ y sólo hay una unidad de función multiplicar, el sumador no se usará durante la primera multiplicación (así como en la fase de arranque de la segunda multiplicación). La contención de banco de memoria es quizá una fuente más importante de degradación de rendimiento. Sin contención de banco de memoria, y suponiendo ancho de banda de memoria adecuado, las operaciones de carga y almacenamiento de registro vectorial se traslapan con cálculos en vectores o

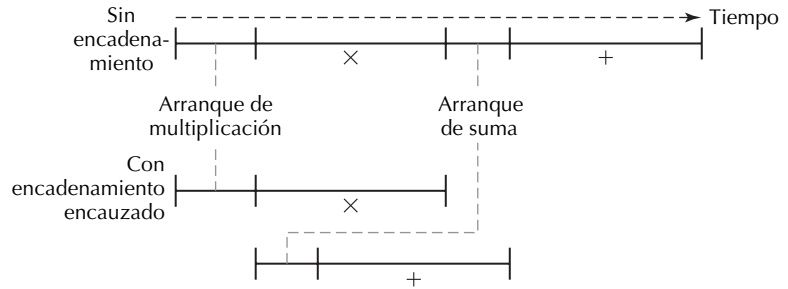


Figura 26.4 Latencia total del cálculo vectorial $S := X \times Y + Z$, sin y con encauzamiento encauzado.

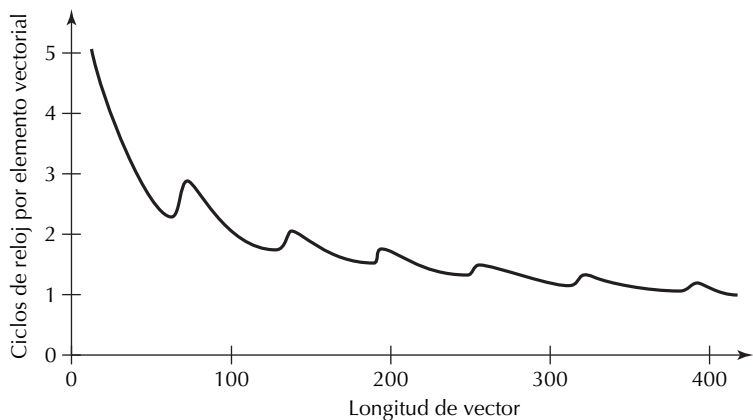


Figura 26.5 El tiempo de ejecución por elemento en un procesador vectorial como función de la longitud del vector.

segmentos de vectores previamente cargados (figura 26.3). Sin embargo, este modo deseable de operación no siempre se puede mantener, en particular cuando a los elementos de vector se debe acceder en posiciones irregulares especificadas mediante vectores índice.

Este tercer factor resulta de la cabecera de la segmentación vectorial cuando todo el vector no encaja en un registro vectorial. Por lo general, se espera que el rendimiento de un procesador vectorial mejore para vectores más largos. Sin embargo, este mejoramiento no es uniforme y suave (figura 26.5). Por ejemplo, con registros vectoriales de 64 elementos, el procesamiento de vectores con longitud 65 involucra mayor cantidad de cabecera para cuidar el elemento adicional.

En cualquier comparación de los MFLOPS pico de un procesador vectorial con los de un procesador escalar, se debe tomar en cuenta el impacto de los cálculos condicionales. Considere, por ejemplo, un ciclo en el que se realizan dos cálculos diferentes, dependiendo de si se sostiene la condición $A[i] > 0$. Para llevar a cabo tal cálculo “si-entonces-de otro modo” (*if-then-else*) en un procesador vectorial, primero se forma un vector máscara que define los elementos de vector que satisfacen la condición. A continuación, se realiza la parte “entonces”, y se usa la máscara para deshabilitar las operaciones para los elementos que no satisfacen la condición. Finalmente, la parte “de otro modo” se realiza para los elementos de vector restantes. Si se supone que los cálculos para las partes “entonces” y “de otro modo” toma la misma cantidad de tiempo e ignora el tiempo necesario para formar el vector

máscara, el tiempo de cálculo casi se duplica y la ventaja de rendimiento del procesamiento vectorial se reduce a la mitad. Puesto de otra forma, la longitud de vector de medio rendimiento crece como resultado de la cabecera para los cálculos condicionales. La penalización de rendimiento de un enunciado “caso” multivía es mayor que un enunciado “si-entonces-sino” de dos vías.

Se han usado muchos métodos arquitectónicos para mejorar aún más el rendimiento de los procesadores vectoriales y mitigar los efectos de factores inhibidores subrayados hasta el momento en esta sección. Uno de tales métodos, el diseño de *procesador vectorial multivía*, permite paralelismo en operaciones vectoriales. Por ejemplo, un diseño de cuatro vías puede dividir cada vector entre cuatro vías, con elementos cuyos índices son congruentes con el módulo 4 asignados a la misma vía. De este modo, por ejemplo, la vía 0 contendrá los elementos 0, 4, 8, . . . , 56, 60 de un vector de 64 elementos que se divide entre cuatro subregistros de 16 elementos (uno por vía). Para realizar una instrucción vector *add* (suma vectorial), se emplean sumadores encauzados para operar en los subregistros en paralelo. Este tipo de paralelismo está arriba y más allá del paralelismo en cuanto tiene múltiples procesadores vectoriales interconectados que usan los métodos discutidos en los capítulos 27 y 28. Para usar una analogía, se pueden ligar múltiples procesadores vectoriales interconectados a múltiples autopistas (*highways*) que permiten que fluya más tránsito desde el punto A hasta el punto B; el diseño multivía es comparable a aumentar el número de vías en una sola autopista; los automóviles que se siguen unos a otros en una vía representan *pipelining* simple. Un lado del beneficio del diseño multivía es que los archivos de subregistro más pequeños para cada vía son más rápidos que un archivo de registro largo.

Otro método arquitectónico es eliminar la mayoría o toda la cabecera de arranque en algunas operaciones vectoriales al permitir la reutilización de *pipelines* de unidad de función sin drenarlas al final de cada instrucción vectorial. En la práctica, es necesario permitir uno o dos ciclos de separación o *tiempos muertos* entre usos consecutivos de la misma *pipeline* de unidad de función, para asegurar que los circuitos de control no se vuelven demasiado complejos. El impacto de rendimiento de este método es particularmente pronunciado cuando los cálculos involucran vectores cortos.

■ 26.4 Sistemas de control compartido

La operación vectorial $Z := X + Y$ constituye una especificación abstracta para m sumas independientes, donde m es la longitud del vector. Esta abstracción se conoce como *computación de datos paralelos*. En lo concerniente a la semántica del programa que abarca esta operación vectorial, no importa si las m sumas independientes se realizan en un procesador escalar una tras otra, por un sumador encauzado profundamente en un procesador vectorial convencional, por medio de cuatro sumadores en un procesador vectorial multivía o mediante m sumadores dentro de m procesadores simples. El último enfoque de usar un número muy grande de procesadores que están centralmente dirigidos para ejecutar operaciones comunes que es un ejemplo de arquitectura de *control compartido*. La noción de control compartido abarca un amplio espectro de opciones, desde compartición completa del control en un ordenamiento SIMD puro (figura 26.6a) hasta, pero no incluyendo, separación total de control o MIMD (figura 26.6c). Los esquemas intermedios, en los que menos que p unidades de control se comparten por p procesadores, se conocen como SIMD o MSIMD, múltiple (figura 26.6b). La asociación de una unidad de control con un subconjunto de p procesadores se puede cambiar estática o dinámicamente, dependiendo de la carga de trabajo. En el resto de este capítulo, el enfoque sobre arquitecturas de control compartido (SIMD) simple también se conoce como *procesadores matriciales*.

En un procesador matricial, la memoria centralizada y el procesador de un procesador vectorial se distribuyen en el espacio, mientras que su control permanece centralizado. Una ventaja inmediata es que el ancho de banda de memoria deja de ser un problema importante. Un vector de longitud m se

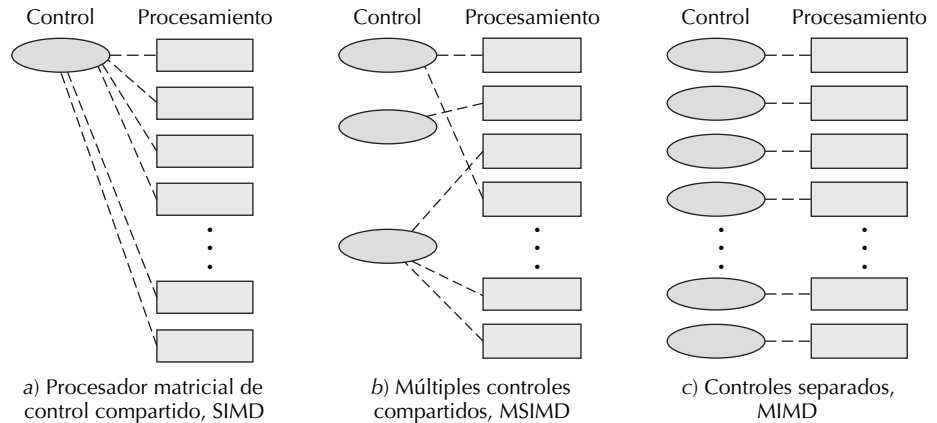


Figura 26.6 De control completamente compartido a controles totalmente separados.

puede distribuir a las memorias locales de los p procesadores, y cada uno recibe aproximadamente m/p elementos de vector. Entonces, una operación vectorial como $Z := X + Y$ se ejecuta rápidamente en el tiempo requerido para m/p sumas. Mientras más grande sea el número de procesadores, menos crítica es la rapidez de la suma en cada uno. Las implementaciones reales de los procesadores matriciales abarcan un amplio rango en la negociación de multiplicidad frente a rendimiento del procesador: desde un número pequeño de los procesadores de alto rendimiento hasta muchos miles de procesadores de bit serial. Un ejemplo del tipo del primer tipo es la ILLIAC IV, una supercomputadora de la década de 1960, que usaba 65-256 procesadores de 64 bits muy rápidos (en su tiempo). El último extremo se ejemplifica mediante las computadoras de máquina de conexión de mediados y finales de la década de 1980, que tenían hasta 64K procesadores de un solo bit, 16 para un chip VLSI. Con la tecnología actual, los sistemas de multimillones de procesadores del último tipo son factibles (256 tabletas \times 64 chips por tableta \times 64-256 procesadores por chip).

El inconveniente de la distribución anterior de las unidades de memoria y procesamiento en el espacio es que se necesita alguna forma de comunicación interprocesador para operaciones vectoriales que involucren combinación (por ejemplo, $s := \sum X$) o para vectores que se procesan en un orden distinto al usado para distribución espacial. Un ejemplo de la última categoría consiste en realizar m multiplicaciones de la forma $X[i] \times Y[m - 1 - i]$ para vectores X y Y de longitud m que se distribuyan de acuerdo con el índice i . En este caso, el procesador que contiene $X[i]$ puede requerir acceso a datos no locales para obtener $Y[m - 1 - i]$.

El patrón de conectividad para la comunicación interprocesado puede variar desde el arreglo lineal muy simple (o anillo), donde cada procesador está conectado a dos procesadores vecinos, hasta redes enormemente sofisticadas a discutirse en la sección 28.2 en conexión con sistemas de multicomputadoras. Un esquema de interconexión bastante natural para los procesadores matriciales es una malla 2D cuadrada o rectangular, que encaja bastante bien con las operaciones de tipo matriz comúnmente ejecutadas en tales sistemas. Por ejemplo, las matrices de 256×256 se pueden procesar eficientemente mediante el procesador matricial conectado en malla de 4×4 de la figura 26.7 si las matrices se dividen en bloques de 64×64 , y a cada bloque se le asigna un procesador en la forma obvia. Con frecuencia es más adecuada la interconexión en toro bidimensional, que es la misma que una malla 2D pero con los dos procesadores al final de cada hilera o columna conectados uno con otro (figura 26.7, que también muestra una posible forma de manejar I/O).

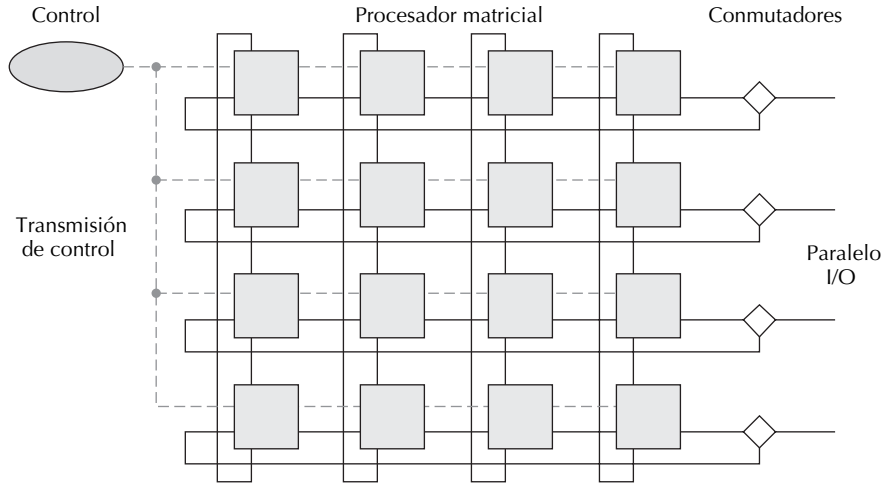


Figura 26.7 Procesador matricial con red de comunicación de interprocesador en toro 2D.

26.5 Implementación de procesador matricial

Los conflictos de implementación para un procesador matricial se discuten en términos del sistema ejemplo que se muestra en la figura 26.7. En otras palabras, se supone que el esquema de interconexión de procesadores en toro 2D se eligió con base en las características de aplicación y que I/O se realizará a través de conmutadores 3×3 que permiten que el arreglo del procesador se conecte como un toro ordinario o para operar en uno de dos modos I/O: flujo de datos dentro de los ciclos de hilera en sentido de las manecillas del reloj o en contrasentido.

La unidad de control de un procesador matricial recuerda un procesador escalar ordinario. Lee (*fetch*) y decodifica instrucciones y transmite (*broadcast*) las señales de control adecuadas a rutas de datos idénticos dentro de todos los procesadores. Cuando el número de procesadores es grande, conectarlos directamente a las señales de control para todos los procesadores puede ser impráctico debido a la gran carga de salida; en tal caso, las señales quizá vayan a través de repetidores. La unidad de control también tiene un archivo de registro, una ALU y una memoria de datos (figura 13.3) para realizar ciertos cálculos relacionados con control y mantener valores escalares que se deben transmitir a los procesadores. El único elemento nuevo que se requiere para la unidad de control, más allá de los de un procesador convencional, es una capacidad para realizar bifurcación con base en el estado del arreglo de procesadores. Esto último requiere que la unidad de control tenga cierto conocimiento global acerca del estado actual de cada procesador. Para este propósito, con frecuencia es adecuado un solo bit de información de cada procesador.

Por ejemplo, después de que al arreglo se le instruye para realizar una prueba (como una comparación aritmética), puede ser interesante determinar si alguno de los procesadores en el arreglo tiene un resultado positivo. La forma más limpia de proporcionar esta capacidad es incluir un *registro de estado del arreglo* en la unidad de control, cuyo ancho es igual al número de procesadores en el arreglo: el bit i en el registro de estado del arreglo se asocia con el procesador i . Entonces, siempre que se emita a los procesadores una operación de prueba condicional, pueden registrar localmente el resultado, así como en su registro de estado. Si después de una prueba, el registro de estado del arreglo contiene 0, entonces la unidad de control sabría que ningún procesador tiene un resultado de prueba exitoso. El número de resultados de este tipo también se puede determinar mediante la unidad de control, si se desea contar

el número de 1 en el registro de estado del arreglo. Para este propósito se puede proporcionar una instrucción especial de *cuenta de población*. Esta capacidad es conveniente en cálculos condicionales “si-entonces-sino”. Si después de probar la condición, localmente en cada procesador, la unidad de control encuentra que ninguna de las pruebas locales fue exitosa, la ejecución de la parte “entonces” se puede saltar. Normalmente, los procesadores que tienen un resultado exitoso se habilitarían para la duración de la ruta “entonces” del programa, y todas las otras participarían en la ejecución de la ruta “de otro modo”.

A los procesadores dentro del arreglo a veces se les denomina procesadores indispensables o *elementos de procesamiento* (PE) para transmitir la falta de algunos de los bloques que se muestran cerca del extremo izquierdo de la figura 13.3 para leer (*fetch*) e interpretar instrucciones. Sin embargo, contienen un archivo de registro, una ALU y una memoria de datos, interconectadas en forma muy parecida a las que se muestran en el lado derecho de la figura 13.3. Una diferencia es que los elementos de procesamiento se pueden apagar selectivamente de modo que ignoren las señales de control transmitidas al arreglo. Esto último resulta necesario para operaciones condicionales. Observe que, aun cuando todos los procesadores activos sigan la misma instrucción, no necesariamente realizan operaciones idénticas sobre los contenidos del mismo registro o localidades de memoria. Se permite cierta autonomía local. Por ejemplo, cuando a la memoria se accede con el uso de la dirección en el registro $\$12$, el contenido del registro $\$12$ puede ser diferente en cada procesador, ello conduce al acceso a diferentes localidades de memoria. Esto facilita los accesos vectoriales de la forma $A[X[i]]$ en el procesador i , donde X es un vector índice.

La figura 26.8 muestra los componentes principales de un elemento de procesamiento, con enfoque particular sobre el método de comunicación interprocesador. La salida de la ALU en cada PE se almacena en un *buffer* de comunicación que es visible a los cuatro PE vecinos en las direcciones norte, este, oeste y sur (abreviado NEWS). En cada ciclo, cuando la comunicación interprocesador se habilita (se postula la señal *CommunEn*), el operando inferior a la ALU se lleva desde un PE vecino, dependiendo de la dirección de comunicación indicada por las señales de control *CommDir*.

Los conmutadores que se muestran cerca del extremo derecho de la figura 26.7 se deben diseñar para permitir tres tipos de conectividad, como se ve en la figura 26.9. Cada conexión bidireccional en la figura 26.7 corresponde a dos ligas unidireccionales en la figura 26.9. En el modo de operación del procesador matricial normal, las conexiones I/O se ignoran y el arreglo funciona como un toro (figura 26.9a). Los dos modos I/O de las figura 26.9b y 26.9c difieren en la dirección de entrada y salida de datos desde la hilera de elementos de procesamiento. Estos dos métodos permiten que los nuevos datos se corran dentro y que los resultados se corran fuera desde los elementos de procesamiento. El mecanismo empleado en el modo operación toro para manejar comunicación interprocesador también permite manejar transferencias de datos I/O que se realizan como operaciones de corrimiento dentro de hileras del arreglo. El diseño del conmutador requerido es directo y se deja como ejercicio.

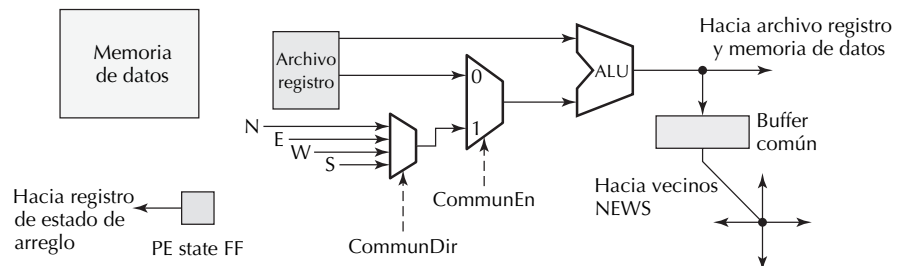


Figura 26.8 Manejo de comunicación interprocesador a través de un mecanismo similar al adelantamiento de datos.

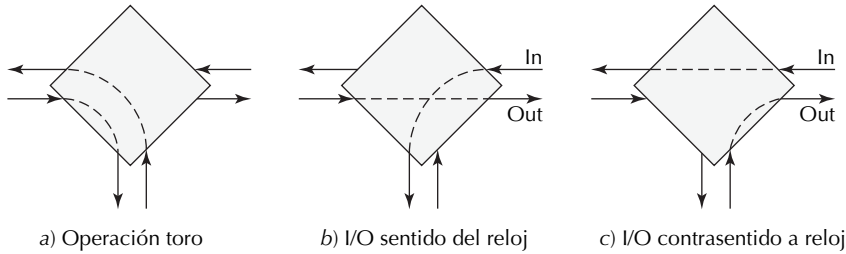


Figura 26.9 I/O conmuta estados en el procesador matricial de la figura 26.7.

26.6 Rendimiento de procesador matricial

Mucho de lo que se dijo en la sección 26.3 acerca del rendimiento del procesador vectorial se aplica también, con muy poco cambio, a los procesadores matriciales. La diferencia entre las dos arquitecturas es que, en un procesador vectorial, un poderoso procesador maneja todos los elementos vectoriales, mientras que en uno matricial cada conjunto de elementos de procesamiento maneja una porción del trabajo. Un procesador vectorial multivía se observa como una combinación de técnicas de procesamiento vectorial y matricial. Para ciertas aplicaciones, un procesador matricial puede ofrecer sustancialmente el mismo rendimiento que uno vectorial con menor costo de diseño y quizá mayor economía de energía. El menor costo de diseño se atribuye a procesadores más simples con optimización de rendimiento menos extensa y problemas de diseño asociados. El reducido consumo de energía se puede deber al uso de circuitos de menor voltaje para implementar PE de bajo rendimiento. Los procesadores matriciales también pueden ser más fuertes y menos proclives a fallas transitorias y de temporización resultantes de frecuencias de reloj muy altas.

Un inconveniente del procesamiento matricial son las cabeceras de transmisión y sincronización. Usualmente, los procesadores corren en paso cerrado, de modo que ejecutan cada instrucción, y notifican sus estados a la unidad de control, al mismo tiempo. La distribución de reloj y su sesgo representan conflictos que deben solucionarse. Usualmente, el tiempo de ciclo de reloj debe aumentar para acomodar tiempo de viaje de señal, así como sesgo de reloj (sección 2.6). La cabecera de comunicación también contribuye a rendimiento reducido, en especial si el esquema de interconexión pide alambres largos entre tarjetas de circuito o gabinetes. Por tanto, si el elemento de procesamiento se usa en la construcción de procesadores de arreglo de tamaños variables, quizá cada procesador tendrá que correr más lento para arreglos más grandes. Por tanto, incluso si el paralelismo de datos es ilimitado, el rendimiento de un procesador matricial no escala linealmente con el número de procesadores.

A las aplicaciones que contienen gran cantidad de paralelismo de datos, y pueden correr eficientemente en un procesador matricial con poca comunicación interprocesador, a veces se les denomina *embarazosamente paralelo* (el autor prefiere usar caracterizaciones más positivas como *agradablemente paralelo*). Los ejemplos de tales aplicaciones abundan en dominios tales como:

- Procesamiento de señal e imagen.
- Gestión de bases de datos y minado de datos.
- Fusión de datos en comando y control.

Como fue el caso para los procesadores vectoriales, las operaciones condicionales en los procesadores matriciales se realizan mediante la ejecución de dos rutas, una tras otra, con el uso de activación selectiva. Lo que era una burbuja en *pipeline* para el caso de un procesador vectorial, se convierte en un elemento de procesamiento inactivo dentro del arreglo que ignora una instrucción particular. Este

desperdicio o ineficiencia debida a recursos inactivos constituye una de las críticas que con frecuencia se expresan acerca del procesamiento matricial. Se puede entender la irrelevancia de este punto al considerar la analogía de un autobús frente a un automóvil particular. El autobús debe seguir una ruta predeterminada, a veces más larga, para asegurar que se atienden todas las áreas de interés para los pasajeros. También puede tener muchos asientos vacíos en algunos viajes. Sin embargo, estos puntos no implican que los autobuses sean modos menos eficientes de transporte; de hecho, para la mayoría de las mediciones son significativamente más eficientes que la alternativa de los automóviles particulares.

Es factible que la unidad de control transmita instrucciones desde ambas rutas de un cálculo condicional de dos vías, donde cada PE siga la instrucción desde la ruta adecuada dependiendo de su estado local. Esto último duplica el número de alambres usados para transmitir y complica los PE porque se necesita un multiplexor para cada señal de control. El rendimiento se puede mejorar porque el tiempo de ejecución de un cálculo condicional de dos vías es dictado por el tiempo de ejecución para la ruta más larga en lugar de la suma de las dos rutas. Llevar este enfoque un paso más allá conduce a la arquitectura SPMD (programa sencillo, datos múltiples) en la que cada procesador tiene su propia unidad de control, pero todos los procesadores ejecutan el mismo programa. De esta forma, cada procesador puede tomar la ruta adecuada dentro de cálculos condicionales de dos o múltiples vías sin que otros procesadores los frenen. Este enfoque también facilita la pesada sincronización puesto que los procesadores requieren estar sincronizados sólo cuando intercambian datos. Este beneficio llega a costa de procesadores más complejos.

Un factor que tiene un impacto significativo sobre el rendimiento de un procesador matricial es la forma en que los datos se distribuyen entre los módulos de memoria asociados con los elementos de procesamiento. Un compilador inteligente con frecuencia puede distribuir los datos en una forma que asegure buen rendimiento. Sin embargo, el problema es inherentemente difícil y no siempre solucionable. Por ejemplo, si una gran matriz distribuye una hilera a cada procesador, entonces los elementos de hilera locales son fácilmente accesibles por un procesador, mientras que la lectura de los elementos particulares de una columna requieren extensa comunicación interprocesador. Más aún, a un arreglo se puede acceder de modo diferente en varias partes de un programa. A veces, paga para reordenar los datos en preparación para un cambio en modo de acceso. La penalización de comunicación pagada durante el reordenamiento puede evitar aún más actividades de comunicación en el curso de cálculos subsecuentes.

PROBLEMAS

26.1 Paralelización de ciclo

Muestre cómo se puede paralelar el ciclo siguiente, discutido en la sección 26.1, a pesar de la obvia dependencia acarreada por el ciclo:

```
for i = 0 to 63 do
  A[i] := A[i] + B[i]
  B[i+1] := C[i] + D[i]
endfor
```

Sugerencia: Invierta el orden de los enunciados.

26.2 Acceso a subarreglos de matriz

Vuelva a hacer el ejemplo 26.1, pero ahora suponga que los elementos de matriz están indexados desde $A[1,1]$ hasta $A[n, n]$; esto es, use indexación origen 1 en lugar de origen 0.

26.3 Duplicación recursiva

Considere el ciclo siguiente para calcular la suma de elementos en un vector X de longitud 64:

```
for i = 0 to 62 do s := s + X[i] endfor
```

La obvia dependencia acarreada por el ciclo evita vectorizar de manera directa este ciclo.

- a) Muestre cómo un compilador vectorizador puede producir código vectorial para este ciclo al sumar pares de elementos consecutivos, luego sumar los resultados desde la primera etapa, etcétera. *Sugerencia:* Use un vector auxiliar Y y forma sumas pareadas de X elementos en $Y[1], Y[3], \dots, Y[63]$, el siguiente conjunto de sumas en $X[3], X[7], \dots, X[63]$, etcétera, y cambia de ida y vuelta entre X y Y ; este método se conoce como *duplicación recursiva*.
- b) Modifique su solución a la parte a) de modo que las *sumas de prefijos* del vector X se calculen en el vector Y , ello significa que, al final, $Y[i]$ contiene la suma de elementos de X , desde $X[0]$ hasta, e incluyendo, $X[i]$.

26.4 Memoria interpolada

Suponga que la unidad de memoria de un procesador vectorial tiene una latencia de 16 ciclos de reloj y que está interpolada en 16 vías. Considere cargar un vector de 64 elementos, con los siguientes pasos, en un registro vectorial. En cada caso, calcule la fracción del ancho de banda de memoria total utilizada y la fracción de ancho de banda relativa a la del paso ideal de la parte a). Compare los resultados y discuta.

- a) $s = 1$
- b) $s = 2$
- c) $s = 4$
- d) $s = 5$
- e) $s = 18$
- f) $s = 20$

26.5 Memoria interpolada

Derive condiciones a satisfacer por el número b de bancos de memoria, el retardo de acceso a memoria d expresado en ciclos de reloj y el paso de acceso a vector s si se deben evitar conflictos de banco durante una operación de carga o almacenamiento vectorial.

26.6 Ciclos con operaciones condicionales

Considere el ciclo siguiente

```
for i = 0 to 63 do if Y[i] ≠ 0 then
    X[i] := X[i]/Y[i] endfor
```

que no se puede vectorizar directamente debido a la operación condicional dentro del ciclo.

- a) Muestre cómo se puede convertir este ciclo a forma vectorial con el uso de operaciones enmascaradas. Un división de vector enmascarado, por ejemplo, lo realiza el elemento i del vector sólo si el bit correspondiente en el vector máscara M es 0; esto es, el elemento no está enmascarado.
- b) Muestre cómo se pueden lograr resultados similares a los de la parte a) sin capacidad de enmascaramiento. *Sugerencia:* Divida $X[i]$ entre $Y[i] + M[i]$, donde $M[i]$ está en $[0,1]$.

26.7 Rendimiento de procesador vectorial

Considere en el ejemplo 26.2 que las operaciones vectoriales caen en dos categorías: las operaciones sobre vectores relativamente cortos no se aceleran en el procesador vectorial debido a cabeceras de arranque y drenado de *pipeline*, mientras que las operaciones con vectores largos gozan de un factor de aceleración de ocho en promedio.

- a) ¿Cuál es la aceleración esperada del procesador vectorial sobre el superescalar si las operaciones de vector corto representan 30% del tiempo de corrido total en el procesador superescalar?
- b) Si supone que no se conocen las fracciones de tiempo de ejecución para operaciones escalar y de vector corto, ¿bajo qué condiciones es alcanzable una aceleración global de 2?
- c) ¿Qué piensa acerca de la efectividad en costo del enfoque de procesamiento vectorial bajo las condiciones de las partes a) y b)? Discuta.

26.8 Rendimiento de procesador vectorial

Otra útil métrica de rendimiento para un procesador vectorial es la longitud de vector necesaria para hacer que el modo vectorial sea más rápido que la operación sobre elementos de vector individuales en modo escalar. Discuta cómo se relaciona esta métrica con la longitud de vector de medio rendimiento y el tiempo de arranque. Establezca claramente todas sus suposiciones.

26.9 Calendarización de carga en registro vectorial

Suponga que un procesador vectorial tiene un gran conjunto de registros vectoriales. Existen dos unidades de

carga que comparten la hilera de solicitudes de carga pendiente que se emitieron en anticipación de futuras operaciones vectoriales. Cada operación carga se caracteriza por una dirección de inicio y un paso. Muestre que puede ser útil realizar las operaciones de carga fuera de orden y discuta los criterios a utilizar para elegir el orden de las operaciones de carga.

26.10 Diseño de elemento de procesamiento

Complete el diseño del elemento de procesamiento que se muestra en la figura 26.8, si supone un conjunto de instrucciones similar al de MicroMIPS dado en la tabla 13.1 (excepto que no hay instrucciones de transferencia de control). Suponga que cada una de las instrucciones se puede condicionar en el estado PE, y agregue un pequeño número de instrucciones (no más de cinco) que permitirían el establecimiento y restablecimiento del flip-flop de estado de PE y use las capacidades de comunicación interprocesador.

26.11 I/O de procesador matricial

- Diseñe el conmutador 3×3 de tres estados de la figura 26.9 de modo que el estado adecuado se supone con base en dos señales de control proporcionadas por la unidad de control compartida.
- Muestre cómo se simplifica el conmutador si se remueve una de las dos direcciones I/O (por ejemplo, en contrasentido de las manecillas del reloj).
- ¿Cómo se diferencia I/O con los conmutadores de las partes a) y b)? En otras palabras, la flexibilidad adicional ofrecida por los conmutadores más complejos de la parte a), ¿valen el costo agregado? Discuta.

26.12 Procesadores asociativos

Un procesador asociativo constituye un procesador matricial en el que los elementos de procesamiento son en extremo simples. Cada PE tiene una sola palabra de memoria (256 bits de ancho) y pueden ejecutar las operaciones siguientes: establecer o restablecer el flip-flop de estado PE o cargarlo con el contenido de un bit especificado en la palabra memoria; realizar varias operaciones lógicas (por decir, AND, NAND, OR, NOR, XOR, XNOR) en dos bits específicos de la palabra de memoria, colocar el resultado en otro bit especificado. Usualmente, la mayoría de la palabra almacenada contiene datos en varios campos, con pocos bits reservados

para retener resultados desde cero conforme proceda el cálculo. Por ejemplo, la palabra almacenada en un PE puede contener datos acerca de un avión que vuela en un espacio aéreo vigilado, con campos de datos correspondientes a ID de avión, sus coordenadas y rapidez. Recuerde que la unidad de control tiene acceso a los bits de estado PE y puede tratar estos bits como en contenido de un registro. Diseñe algoritmos de procesador asociativo para los siguientes cálculos, en cada caso almacenando los resultados de bit sencillo en flip-flops de estado PE. Suponga que el campo de interés F contiene un entero sin signo y se extiende desde el bit i hasta el bit j , donde $j > i$.

- Determine cuáles PE contienen un valor distinto de cero en el campo F .
- Determine si algún PE contiene el valor 23 en el campo F .
- Determine si algún PE contiene un valor mayor que o igual a 18 en el campo F .
- Determine cuáles PE contienen un valor entre 18 y 25 en el campo F .
- Determine cuáles PE contienen el valor más grande en el campo F .

26.13 Búsquedas de membresías

En el problema 26.12 se le pidió mostrar cómo se realizan algunas búsquedas simples en un procesador asociativo genérico. Este problema trata con *búsquedas de membresía* en procesadores asociativos que usan las mismas suposiciones que en el problema 26.12.

- Muestre cómo se pueden identificar todos los elementos de procesamiento que contienen uno de los patrones de bit 0101, 0110, 0111, 1101, 1110 o 1111 en un campo particular de cuatro bits, con las menos instrucciones posibles.
- Formule un procedimiento general para realizar búsquedas de membresía del tipo dado en la parte a) con un número mínimo de instrucciones.

26.14 Juego de la vida de Conway

Diseñe una máquina SIMD simple para jugar el Juego de la Vida de John Conway. En este juego, el mundo de los microorganismos está modelado por una matriz booleana, donde 1 representa la presencia y 0 la ausencia de un organismo viviente. Se supone tiempo discreto

y el nuevo estado de cada celda de matriz en el tiempo $t + 1$ se determina mediante cuatro reglas basadas en el número de organismos vivientes en sus ocho vecinos más cercanos en el tiempo t : 1) Cualquier organismo viviente con dos o tres vecinos, sobrevive. 2) cualquier organismo viviente con cuatro o más vecinos, muere o está apiñado. 3) Cualquier organismo viviente con cero o un vecino, muere de soledad. 4) Un organismo nace en cualquier celda vacía con exactamente tres vecinos. Su máquina debe ser capaz de simular el Juego de la Vida en una matriz de 256×256 durante muchos millones de pasos de tiempo (generaciones).

26.15 Procesamiento vectorial frente a matricial

- a) ¿El efecto “cascada” que se muestra en la figura 26.5, en conexión con los procesadores vectoriales, se aplica también a procesadores matriciales? ¿En qué forma o por qué no?
- b) ¿Qué características de una aplicación o sus datos de entrada/salida dictan si el procesamiento vectorial o matricial será adecuado o más efectivo en costo para él?

26.16 Acelerador SIMD para procesamiento de imagen

Dos tipos de operación se usan generalmente en el procesamiento de imagen. Suponga que una imagen se representa mediante una matriz 2D de 0 y 1 que corresponden a píxeles oscuros y claros, respectivamente. La remoción de ruido tiene la meta de remover 0 y 1 aislados (por decir, aquellos que tienen cuando mucho un pixel del mismo tipo entre sus ocho vecinos adyacentes horizontal, vertical o diagonalmente). El suavizado se realiza al sustituir cada valor de pixel con la mediana de nueve valores que consisten del pixel mismo y sus ocho vecinos.

- a) Discuta el diseño de una unidad SIMD compuesta de un arreglo 2D de celdas simples para acelerar la remoción de ruido o el suavizado.
- b) Proponga generalizaciones adecuadas para remoción de ruido y suavizado si cada valor de pixel es un entero binario (por decir, de cuatro bits de ancho) que representan un nivel de gris.
- c) Bosqueje las modificaciones requeridas para las celdas de la parte a) si se va a implementar la parte b).

REFERENCIAS Y LECTURAS SUGERIDAS

- | | |
|----------|---|
| [Asan03] | Asanovic, K., “Vector Processors”, Apéndice G de <i>Computer Architecture: A Quantitative Approach</i> , por J. L. Hennessy y D. A. Patterson, Morgan Kaufmann, 3a. ed., 2003 (disponible en línea en www.mkp.com/CA3/). |
| [Crag96] | Cragon, H. G., <i>Memory Systems and Pipelined Processors</i> , Jones and Bartlett, 1996. |
| [Hord90] | Hord, R. M., <i>Parallel Supercomputing in SIMD Architectures</i> , CRC Press, 1990. |
| [Parh95] | Parhami, B., “SIMD Machines: Do They Have a Significant Future?”, <i>ACM Computer Architecture News</i> , vol. 23, núm. 4, pp. 19-22, septiembre de 1995 (versión más corta en <i>IEEE Computer</i> , vol. 28, núm. 6, pp. 89-91, junio de 1995). |
| [Thom93] | Thomborson, C. D., “Does Your Workstation Computation Belong on a Vector Supercomputer?”, <i>Communications of the ACM</i> , vol. 36, núm. 11, pp. 41-49, 94, noviembre de 1993. |

■ CAPÍTULO 27

MULTIPROCESAMIENTO DE MEMORIA COMPARTIDA

“Un amigo en la vida es mucho; dos son demasiados; tres difícilmente es posible. La amistad requiere cierto paralelismo de vida, una comunidad de pensamiento, una rivalidad de metas.”

Henry Brooks Adams

“Los avances en tecnología tanto en hardware como en software han puesto a debate la mayoría de los conflictos en arquitectura de conjunto de instrucciones... El diseño de computadoras se enfoca más en los problemas de ancho de banda de memoria e interconexión de multiprocesamiento.”

Robert Tomasulo, Viendo hacia adelante, circa 1998

TEMAS DEL CAPÍTULO

- 27.1** Memoria compartida centralizada
- 27.2** Cachés múltiples y coherencia de caché
- 27.3** Implementación de multiprocesadores simétricos
- 27.4** Memoria compartida distribuida
- 27.5** Directorios para guía de acceso a datos
- 27.6** Implementación de multiprocesadores asimétricos

La idea de usar múltiples procesadores para cooperar en un problema computacionalmente intenso mediante el acceso a estructuras de datos comunes en una memoria compartida, donde cada uno contribuya a una solución global, parece ingenua al principio. ¿No se llegó a la conclusión de que, de hecho, la rapidez de la memoria, no la del procesador, es la que usualmente limita el rendimiento? ¿Cómo, entonces, tiene sentido esperar que una sola unidad de memoria continúe con muchos procesadores? Responder esta pregunta es la meta de este capítulo. Por ejemplo, observe que al proporcionar una caché privada para cada procesador, se puede reducir la frecuencia de accesos a la memoria compartida común. Sin embargo, esta provisión conduce al desafiante problema de garantizar consistencia entre los datos en múltiples caché. Se verá cómo este y otros problemas se solucionan en muchas clases de multiprocesadores.

■ 27.1 Memoria compartida centralizada

Los módulos de memoria que comparten algunos procesadores pueden estar localizados centralmente o distribuidos y empacados con los procesadores. En el primer caso, todo el espacio de dirección en

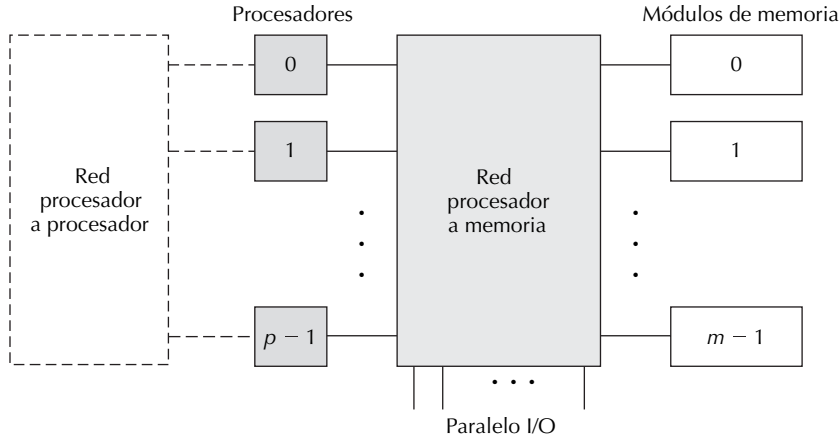


Figura 27.1 Estructura de un multiprocesador con memoria compartida centralizada.

memoria es igualmente accesible a todos los procesadores, ello conduce a arquitecturas de *multiprocesadores simétricos* o *acceso a memoria uniforme* (UMA = *uniform memory access*). En el último caso, la *memoria local* es más fácilmente accesible a un procesador que la *memoria remota* ubicada con otros procesadores. La memoria compartida distribuida conduce a arquitecturas de *multiprocesadores asimétricos* o *acceso a memoria no uniforme* (NUMA = *nonuniform memory access*). En esta sección se discute la memoria compartida centralizada, la consideración de la memoria compartida distribuida se deja para la sección 27.4.

La figura 27.1 muestra la estructura general de un multiprocesador con memoria compartida centralizada. Los procesadores pueden acceder a la memoria compartida a través de una red de interconexión procesador a memoria que permite que cualquier procesador lea o escriba datos desde/hacia cualquier módulo de memoria. La más simple de tales redes constituye un bus compartido al que están conectados todos los procesadores y módulos de memoria. Usar un bus compartido como la red de interconexión no es viable en la estructura que se muestra en la figura 27.1, aunque puede serlo en cuanto se agregan cachés (sección 27.2). En el otro extremo, se puede usar una red de intercambio que permita que todos los procesadores accedan simultáneamente a distintos módulos de memoria. La más práctica red de interconexión procesador a memoria cae entre estos dos extremos. También puede haber una red de interconexión interprocesador separada, usualmente dedicada al paso de datos e información de control para coordinación y sincronización (como las señales de interrupción) entre los procesadores.

La red de interconexión procesador a memoria representa el componente más crucial en el multiprocesador de memoria compartida de la figura 27.1. Debe ser capaz de soportar accesos a memoria libres de conflicto y alta rapidez mediante muchos procesadores a la vez. Aun cuando la latencia de acceso a memoria a través de la red de interconexión es muy importante, el ancho de banda de comunicación que se puede soportar es todavía más importante, pues se pueden usar el multihilo y otros métodos para permitir a los procesadores continuar realizando trabajo útil mientras se espera que lleguen desde memoria los datos solicitados. Se dice que estos métodos proporcionan *tolerancia a latencia* u *ocultamiento de latencia*.

Con el propósito de proporcionar una red de interconexión con múltiples rutas y ancho de banda adecuado para soportar accesos a memoria simultáneos desde muchos procesadores, con frecuencia se usa gran número de conmutadores (*switches*) ordenados en capas. La resultante *red de interconexión multietapa* puede seguir muchas estrategias de diseño diferentes, lo que conduce a variadas comple-

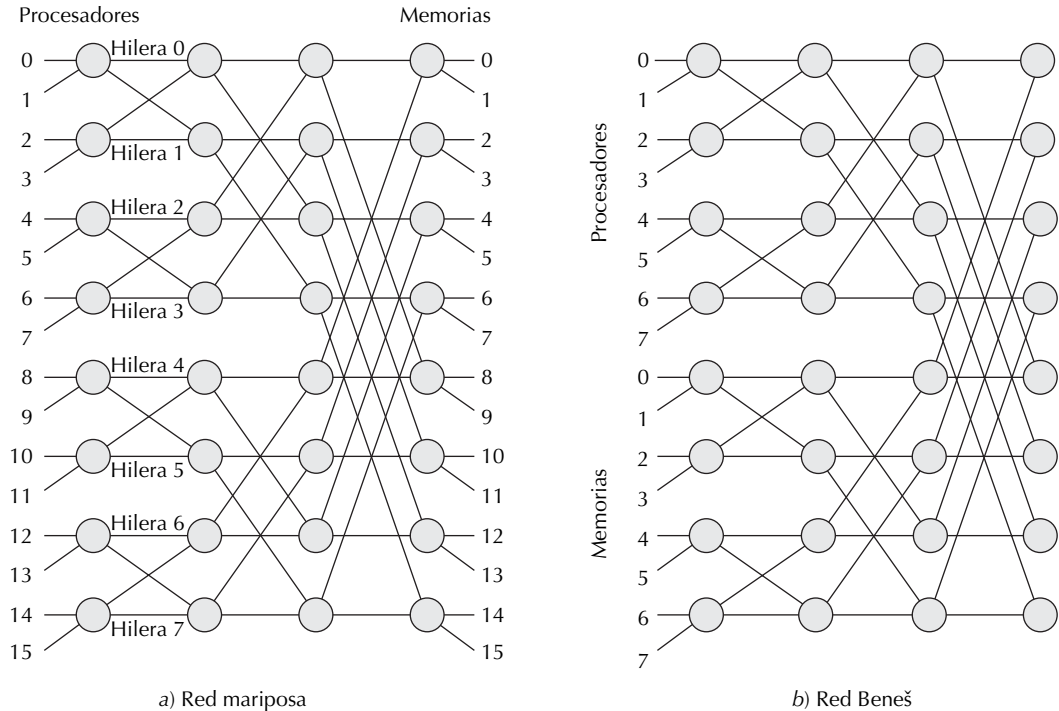


Figura 27.2 Redes de mariposa y Beneš relacionadas como ejemplos de redes de interconexión procesador a memoria en multiprocesadores.

jidades y rendimientos totales. Los diseños disponibles ofrecen un amplio espectro de opciones en términos de negociaciones costo/rendimiento. La teoría de redes de interconexión es muy rica, todavía se están descubriendo nuevas estructuras que ofrecen ciertas ventajas en términos de facilidad de implementación o rendimiento [Parh99].

Una de las más viejas y más versátiles estructuras de interconexión multinivel es la *red mariposa*, que consiste de *switches* ordenados en 2^q hileras y $q + 1$ columnas. Si se procede de izquierda a derecha, cada *switch* en la hilera r tiene una conexión directa con un *switch* en la misma hilera y una conexión cruzada con *switch* en la hilera 2 ± 2^c , donde c es el número de columna. La figura 27.2a muestra cómo se puede usar una red mariposa de ocho hileras para conectar 16 procesadores a 16 módulos de memoria. Cualquier procesador i se puede conectar a cualquier módulo de memoria j . Más aún, hay una ruta única para esta conexión, que se puede determinar fácilmente a partir de las representaciones binarias de i y j . Por ejemplo, la ruta desde el procesador 5 hasta el módulo de memoria 9 se determina al observar primero que la ruta conduce de la hilera 2 = $(010)_{\text{dos}}$ de la mariposa a la hilera 4 = $(100)_{\text{dos}}$. Al operar XOR 010 y 100 produce la *etiqueta de enrutamiento* (routing tag) 110, que se puede usar para rastrear una ruta desde la hilera 2 a la hilera 4 de la red mariposa. Un mensaje que entra a un *switch* puede dejarlo al tomar la ruta directa o la cruzada. La lectura hacia atrás de la etiqueta de enrutamiento produce la ruta. En el ejemplo, se obtiene la ruta 0 (directa), 1 (cruzada), 1 (cruzada), que fácilmente se verifica que conduce desde la hilera 2 hasta la hilera 4. Un mensaje que porta la etiqueta de enrutamiento puede encontrar su camino a través de la red sin necesidad de control externo. Por esta razón, la red mariposa es un ejemplo de *red de autoenrutamiento*.

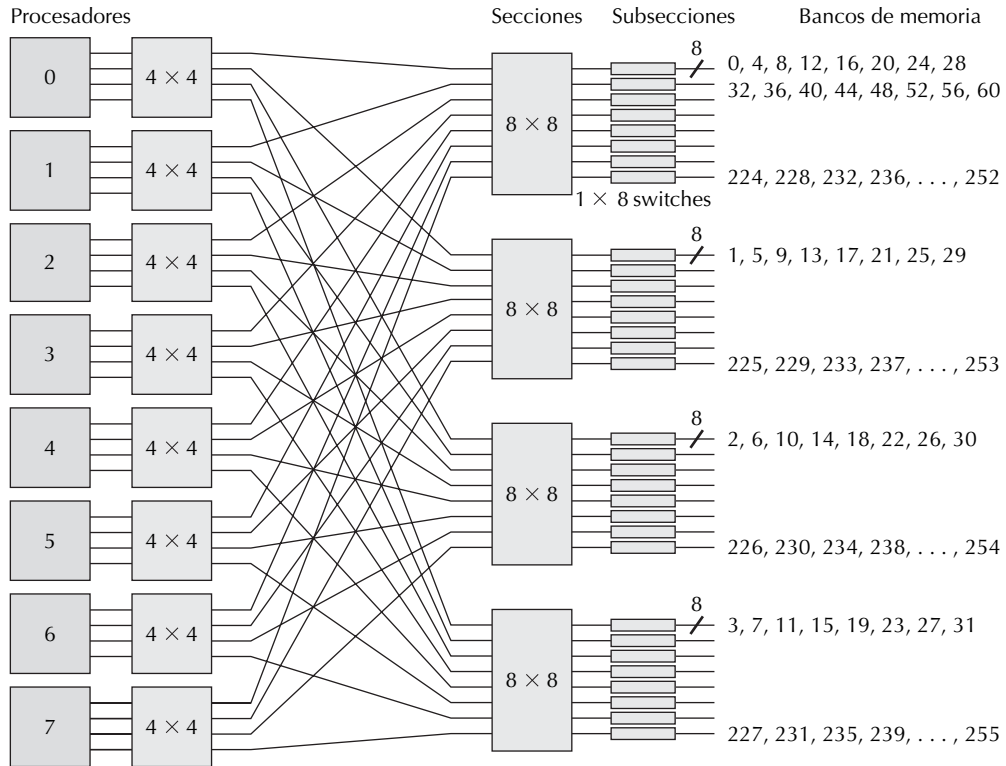


Figura 27.3 Interconexión de ocho procesadores a 256 bancos de memoria en Cray Y-MP, una supercomputadora con múltiples procesadores vectoriales.

La red mariposa es capaz de conectar muchos procesadores en un lado a los módulos de memoria en el otro; sin embargo, no puede establecer conexiones de acuerdo con un intercambio arbitrario. En otras palabras, no es una red de intercambio. Un tipo particular de red de intercambio $p \times q$, conocida como red Beneš, se puede derivar a partir de una mariposa de p hileras (figura 27.2b). Esta red tiene algunas redundancias en cuanto a que los *switches* en su columna de la extrema derecha no realizan alguna función útil. Se ve fácilmente que la red se puede simplificar a dos redes mariposa de 4 hilos espalda con espalda. Los detalles se dejan como ejercicio.

Las redes que se muestran en la figura 27.2 usan *switches* 2×2 . Se pueden emplear *switches* de otros tipos para construir redes más sofisticadas, para reducir la latencia de acceso a memoria o para acomodar diferente número de procesadores y módulos de memoria. Por ejemplo, la figura 27.3 muestra la red Cray Y-MP usada para conectar ocho procesadores vectoriales a 256 bancos de memoria. Está compuesta de *switches* 4×4 , 8×8 y 1×8 en tres capas.

El modelo de programación de memoria compartida es intuitivo, esto constituye una de las ventajas del multiprocesamiento con memoria compartida. Una abstracción de una computadora con memoria compartida, conocida como máquina de acceso aleatorio paralelo (PRAM, por sus siglas en inglés), permite la especificación de accesos lectura y escritura a cálculo y memoria para cada uno de los p procesadores en cada ciclo de operación. Por ejemplo, si supone un vector B de p elementos en la memoria compartida, el siguiente programa copia el valor en $B[0]$ en todos los otros elementos vectoriales; esta es una forma de transmisión en cuanto a que posteriormente permite que todos los p procesadores ga-

nan acceso al valor común a través de accesos a memoria libres de conflicto, siempre que los elementos de B se distribuyan entre diferentes módulos de memoria.

```
for k = 0 to  $\lceil \log_2 p \rceil - 1$  processor j,  $0 \leq j < p$ , do
    B[j +  $2^k$ ] := B[j]
endfor
```

Observe que, en cada paso, los p procesadores leen desde y escriben en diferentes localidades de memoria. Un algoritmo similar se puede usar para determinar la suma de todos los elementos en un vector X . Aun cuando el siguiente algoritmo parezca diferente del precedente, sigue fundamentalmente la misma lógica. La determinación de las diferencias en la apariencia de los dos algoritmos, y la especificación de las razones para ellas, se deja como ejercicio.

```
processor j,  $0 \leq j < p$ , do Z[j] := X[j]
s := 1
while s < p processor j,  $0 \leq j < p - s$ , do
    Z[j + s] := X[j] + X[j + s]
    s := 2 * s
endfor
```

El punto de estos ejemplos es mostrar que, aparte de tener procesadores múltiples para los que se deba especificar un cálculo en cada paso, la programación de los multiprocesadores con memoria compartida es intuitiva y conceptualmente similar a la programación ordinaria.

■ 27.2 Cachés múltiples y coherencia de caché

La organización de multiprocesador de la figura 27.1 presenta dos problemas principales. Primero, el alto volumen de datos entre procesadores y módulos de memoria hace que la red de interconexión procesador a memoria sea crucial y quizá se convierta en un problema de rendimiento a menos que se diseñe para un ancho de banda extremadamente alto; esto hace a la red muy compleja y, por tanto, costosa. Segundo, incluso si la latencia de la red de interconexión y el ancho de banda no son factores limitantes, la contención entre procesadores para el acceso a los módulos de memoria puede frenar los accesos significativamente y reducir la aceleración efectiva debido a procesamiento paralelo. Por ejemplo, si existen piezas de datos compartidos a los que continuamente acceden muchos procesadores, el ancho de banda de memoria efectivo será mucho menor que el máximo ofrecido por m módulos de memoria. A tales piezas de datos compartidos se les refiere como puntos calientes (*hot spots*) y a los correspondientes conflictos de acceso como *contención de punto caliente*. La contención de acceso a memoria crea una seria barrera para el rendimiento, en particular si considera que la memoria es un recurso limitante incluso con un solo procesador (vea “golpear la pared de memoria” en la sección 17.3).

Una posible solución para los dos problemas anteriores consiste en reducir la frecuencia de accesos a memoria principal mediante el equipamiento de cada procesador con una memoria caché privada. Esto funcionaría perfectamente si no fuera por el requisito de compartir datos entre procesadores para permitirles resolver problemas cooperativamente. Una tasa de impacto de 95% en las cachés reduciría el tráfico a través de la red procesador a memoria a 5% de la de sin cachés, ello mitiga el impacto de las latencias de red y acceso a memoria, así como la posibilidad e impacto de conflictos de acceso a memoria. Para entender algunas de las complejidades creadas por los datos compartidos ante las múltiples cachés, considere las situaciones de la figura 27.4, donde se muestran cuatro líneas de caché: w , x , y y z . La línea w se copió en dos cachés, y las dos copias son consistentes una con otra y con la que está en la memoria principal. En tanto a las dos copias de w se acceda en modo sólo lectura, el estatus

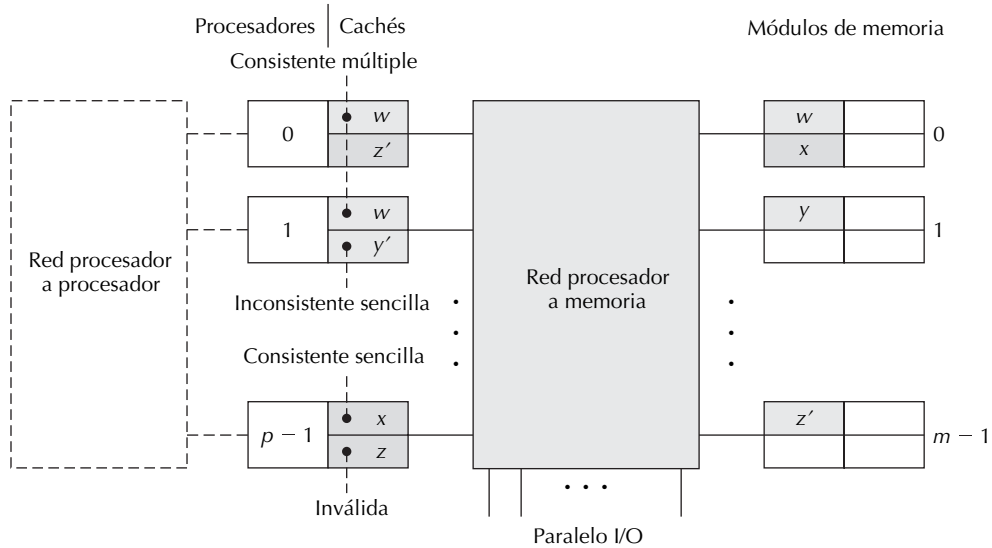


Figura 27.4 Bloques de datos con caché en un procesador paralelo con memoria principal centralizada y cachés de procesador privados.

de línea permanece “consistente múltiple”. Por tanto, múltiples copias de datos sólo lectura en los diversos cachés no presentan problema. De igual modo, una sola copia en caché de una línea particular, ya sea que la copia esté sin modificar (limpia) o modificada (sucía), o es problemático en tanto la línea modificada se reescriba a memoria principal cuando se sustituya.

Sin embargo, si se permite a los procesadores escribir en sus líneas de datos en caché, se pueden desarrollar inconsistencias entre las múltiples copias. Las estrategias para asegurar la consistencia de datos se conocen como *algoritmos de coherencia de caché*. Una posible estrategia para asegurar la coherencia de caché es la siguiente. Designe cada línea caché como exclusiva o compartida. Una línea exclusiva es la que sólo está en una caché; se dice que tal línea está apropiada por el caché o el procesador asociado. Un procesador puede leer libremente desde una línea de caché compartida o exclusiva, pero sólo puede modificar una línea de caché exclusiva. La modificación de una línea compartida requiere que su estatus se cambie a exclusivo mediante la invalidación de todas las copias de dicha línea en otras cachés. En otras palabras, la intención para modificar la línea se debe transmitir a todas las otras cachés de modo que cada caché pueda invalidar la copia que contiene, si hay alguna. Este enfoque es un ejemplo de un algoritmo de coherencia de caché *invalidar escritura*. Sin embargo, son posibles muchos otros algoritmos.

La figura 27.5 representa el *protocolo snoopy* (protocolo entrometido, curioso), una implementación particular del anterior algoritmo de coherencia de caché invalidar-escritura, que supone cachés de reescritura. El protocolo snoopy se apoya en un bus compartido hacia el cual están conectadas todas las unidades caché. Las unidades de control *snoop*, asociadas con cada una de las unidades caché, vigilan continuamente el bus. Cada línea de caché puede estar en uno de tres estados: compartido, exclusivo o inválido. Nombres alternos para los estados compartido y exclusivo son “limpio válido” y “sucio válido”, respectivamente. Las transiciones entre estados ocurren en respuesta a eventos locales (impacto de lectura CPU, fallo de lectura CPU, impacto de escritura CPU, fallo de escritura CPU) y eventos en otras cachés, como se observan en el bus (fallo de lectura de bus, fallo de escritura de bus). Los impactos de lectura de CPU nunca son problemáticos y no conducen a un cambio de estado. Un impacto de

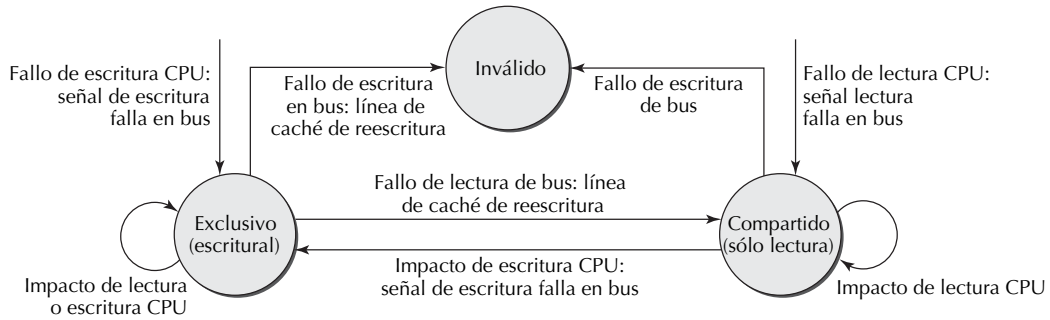


Figura 27.5 Mecanismo de control de estado finito para un protocolo de coherencia de caché *snoopy* basado en bus con cachés de reescritura.

escritura CPU para una línea en un estado exclusivo igualmente no causa problema o modificación. Sin embargo, un impacto de escritura para una línea en estado compartido, hace que el estado cambie a exclusivo y se señale un fallo de escritura a otras cachés, ello las hace invalidar sus copias. Si se observa un fallo de escritura de bus para una línea que está en estado exclusivo, la línea se escribe de vuelta a memoria y el estado cambia a inválido. Un fallo de lectura de bus en estado exclusivo igualmente causa una operación de reescritura, pero el estado cambia a compartido. Finalmente, los fallos de lectura o escritura CPU hacen que el evento se señale en el bus y que la línea de fallo se lleve al caché en estado compartido o exclusivo, respectivamente.

Observe que, siempre que en el bus aparezca una solicitud de acceso a una línea de caché exclusiva, el uso de una política de reescritura fuerza al propietario de dicha línea a intervenir para asegurar que no se usen datos antiguos de la memoria principal. Por tanto, además de reescribir la línea a memoria principal, las transiciones de estados exclusivo a compartido e inválido requieren la inhibición de cualquier acceso a memoria iniciado como resultado del fallo de lectura o escritura (o que se suprima el acceso a memoria, que puede estar en progreso).

El diagrama de estado de la figura 27.5 a veces se dibuja con transiciones adicionales entre los tres estados. Por ejemplo, puede haber una transición del estado exclusivo al compartido denominado “fallo de lectura CPU: reescribir línea caché, señal lectura falla en bus”. ¡Es un poco críptico encontrar un fallo de transición para un estado que indica la disponibilidad de datos en caché! La implicación aquí es que se encuentra un fallo de lectura para alguna línea deseada que se debe colocar donde actualmente no se tiene otra línea en estado exclusivo. Obviamente, en este caso, la línea exclusiva (que es sucia) se debe reescribir a memoria antes de que se pueda sustituir con la línea deseada. Esta nueva línea supone el estado compartido, pues el fallo ocurrió para una operación de lectura. Se eligió dibujar el diagrama en la forma más simple que se muestra en la figura 27.5, para evitar desorden y confusión. Siempre se implica el manejo adecuado de cualquier bloque que se sustituya por otro.

■ 27.3 Implementación de multiprocesadores simétricos

La implementación más común de un *multiprocesador simétrico* (SMP, por sus siglas en inglés) usa un bus compartido para interconectar un número de pequeño a moderado (4-64) de procesadores, cada uno con su propia caché privada; el procesamiento paralelo a gran escala, que implica de cientos a muchos miles de procesadores, es impráctico con este método. En esta sección el foco está sobre las implementaciones basadas en bus, así como en el ofrecimiento de un cuadro razonablemente completo de los conflictos y negociaciones de la implementación. Se harán algunas suposiciones simplificadoras con la finalidad de reducir la presentación a una longitud apropiada para este libro. El capítulo 6 de

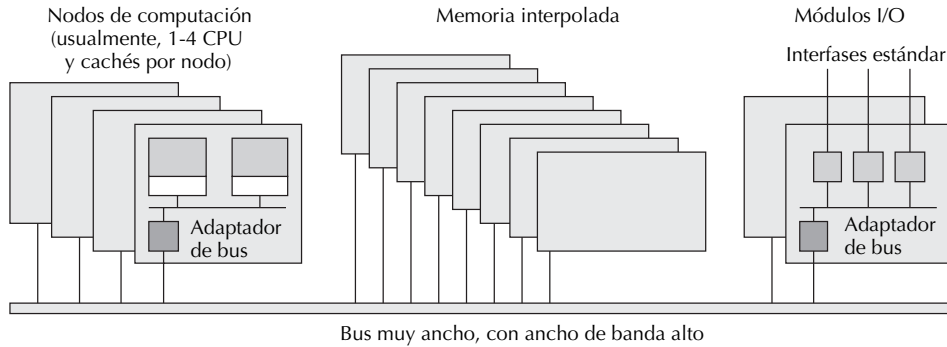


Figura 27.6 Estructura de un multiprocesador simétrico genérico basado en bus.

[Cull99] contiene una discusión mucho más extensa de los conflictos de implementación para multiprocesadores basados en *snoop*, incluidos los efectos de muchos parámetros que no se tratan aquí.

La figura 27.6 muestra la estructura de un multiprocesador simétrico genérico basado en bus. Tres tipos principales de componentes se conectan a un bus con ancho de banda alto, y usualmente muy ancho. El rendimiento del bus es absolutamente crucial en tal sistema, dado que, incluso en un uniprocador de un solo bus, no es raro que el ancho de banda del bus dicte el rendimiento global.

Ejemplo 27.1: El ancho de banda de bus limita el rendimiento Considere un multiprocesador con memoria compartida construido en torno a un solo bus con un ancho de banda de datos de x GB/s. Las palabras de instrucciones y datos usualmente tienen 4 B de ancho, cada instrucción ejecutada requiere acceso a un promedio de 1.4 palabras de memoria (incluida la instrucción misma) y la tasa de impacto combinada para las cachés es de 98%. Calcule una cota superior sobre el rendimiento del multiprocesador en GIPS. Suponga que las direcciones se transmiten sobre líneas de bus separadas y, por tanto, no afectan el ancho de banda de datos del bus.

Solución: Dada la tasa de fallo colectiva de 0.02, la ejecución de una instrucción implica una transferencia de datos de bus de $1.4 \times 0.02 \times 4 = 0.112$ B. Por tanto, una cota superior sobre el rendimiento es $x/0.112 = 8.93x$ GIPS. Si se supone un ancho de bus de 32 B, no se desperdician ciclos de bus o datos, y una tasa de reloj de bus de y GHz, la cota superior de rendimiento se convierte en $286y$ GIPS. Observe que la cota superior así derivada es sumamente optimista, y el rendimiento real puede ser una pequeña fracción de esta cota. Esto último, combinado con el hecho de que los buses de alto rendimiento actualmente operan en el rango de 0.1-1 GHz, implica que un nivel de rendimiento cercano a 1 TIPS (incluso 0.25 TIPS) está más allá del alcance con este tipo de arquitectura, sin que importe el número de procesadores usados.

Como se muestra en la figura 27.6, aparte del bus, usualmente existen tres tipos de componentes en un multiprocesador simétrico de bus compartido. Hay algunos nodos de computación, cada uno contiene 1-4 procesadores más las memorias caché asociadas. Los múltiples procesadores en un nodo de computación están ligados juntos mediante un bus de nodo interno y comparten un puente de bus para conexión al bus de sistema principal. Los módulos de memoria interpolada permiten acceso a memoria de ancho de banda alto. Puede haber 4-16 módulos de memoria de acceso independiente. Puesto que el ciclo de bus es mucho más corto que la latencia de acceso a memoria, el bus puede usar un *protocolo de transacción dividida* por medio del cual el bus se libere entre la transmisión de dirección de memoria

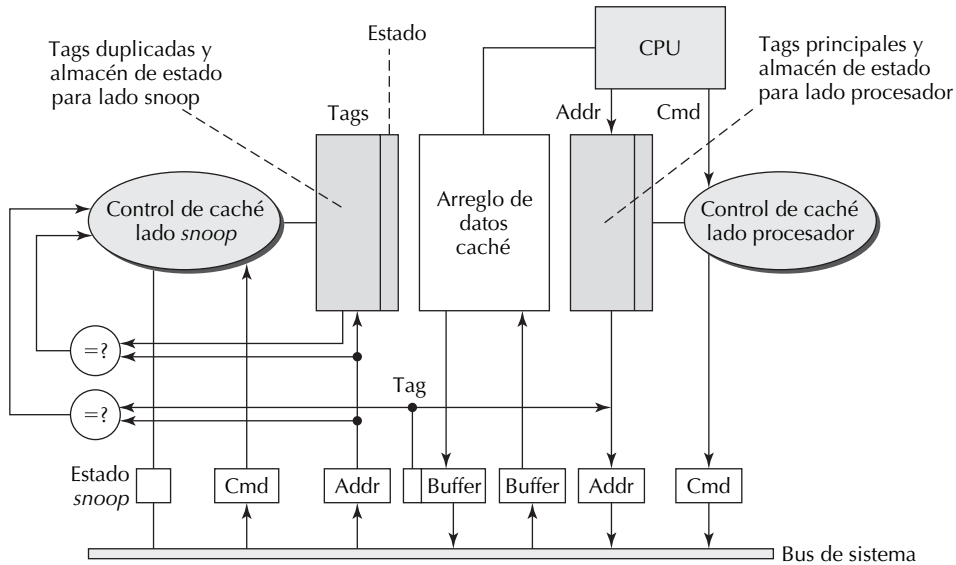


Figura 27.7 Estructura principal para un algoritmo de coherencia de caché basada en *snoop*.

y el retorno de datos. Por tanto, en cualquier momento puede haber en progreso muchas solicitudes de acceso a memoria sobresalientes. Muchos módulos I/O redondean los componentes, cada uno está organizado en torno a un bus I/O que se liga al bus de sistema principal mediante un puente. Para los dispositivos I/O se proporcionan varias interfaces estándar, a través de controladores conectados al bus I/O de cada módulo.

En virtud de que forzar la coherencia de caché es uno de los aspectos clave de dicho multiprocesador, a continuación se discute en forma sumamente simplificada el diseño de una unidad de coherencia basada en *snoop*, al suponer un solo nivel de caché para cada procesador. Con el propósito de simplificar aún más el diseño, se supone un bus atómico en lugar de otro de transacción dividida. Esta última decisión es estrictamente para limitar la complejidad del diseño y no es una buena elección para un sistema práctico. La figura 27.7 muestra los componentes principales en el sistema de control de caché. En virtud de que el *snooping* debe tener lugar continuamente, se duplican las tags y bits de estado para la caché a fin de permitir que el procesador acceda a la caché concurrentemente con el *snooping*. Cualquier actualización a las tags e información de estado se debe, desde luego, realizar en ambas copias. En el lado del procesador, un fallo de caché causa que una dirección y comando se coloquen en el bus, y que los datos regresados se coloquen más tarde en un *buffer* de lectura. Un evento de reescritura hace que los datos y tag asociada se coloquen en un buffer de reescritura para la posterior transmisión en el bus. En el lado *snoop* se examinan el comando y la dirección en el bus, y las comparaciones de tag se realizan tanto con los datos de caché como con el contenido del *buffer* de reescritura.

Puesto que el bus de sistema atómico sencillo puede llevar a cabo una transacción a la vez, se impone un orden total en dichas transacciones, ello significa que todas las cachés las “verán” en el mismo orden. A pesar de estas circunstancias simplificadoras, surgen algunas complicaciones debido a eventos simultáneos en múltiples procesadores y cachés asociadas. Por ejemplo, si dos procesadores emiten solicitudes para escribir en una línea de caché compartida (cambiar su estado de compartida a exclusiva) al mismo tiempo, el árbitro de bus permitirá que una de éstas pase primero. El otro procesador debe entonces invalidar la línea y emitir una solicitud diferente correspondiente a un fallo de escritura de caché. Una forma conveniente de manejar éste y casos similares consiste en incluir estados intermedios

en el diagrama de estado de la figura 27.5. Por ejemplo, la transición de estado compartido a exclusivo no ocurre instantáneamente sino más bien a través de un estado intermedio compartido/exclusivo. La solicitud del procesador en este escenario cambiará el estado de compartido a compartido/exclusivo. Posteriormente, ocurrirá la transición del estado compartido/exclusivo al estado exclusivo o inválido, dependiendo del resultado de la solicitud. Los detalles de la implementación, así como conflictos más sutiles de bloqueo (*deadlock*) y *starvation* (inanición) debido a solicitudes repetidas por múltiples procesadores para la misma línea de caché, están más allá del ámbito de la discusión; se pueden encontrar en otras partes [Cull99].

El diseño de un multiprocesador con memoria compartida también debe proporcionar dos mecanismos útiles. El primer mecanismo permite bloquear los recursos compartidos para uso exclusivo de un procesador. Tal *exclusión mutua* se puede asegurar mediante mecanismos especiales de hardware o a través de software si ciertas instrucciones atómicas se soportan por el hardware. Una instrucción atómica es la que se realiza como un todo, sin la posibilidad de que cambie el contenido de memoria antes de su conclusión. Para ver por qué se necesita tal instrucción, considere el uso de un cierre de software en memoria, donde 0 (1) representa el estado abierto (cerrado). Un procesador que desee cerrar un recurso para uso exclusivo puede leer el contenido del cierre, observar que es 0 y entonces escribir un 1 para adquirir el recurso. Sin embargo, después de leer el cierre, y antes de cambiar su valor con otra instrucción, un segundo procesador puede leer el cierre y presumir que el recurso está disponible para su uso. Entonces ambos procesadores escriben 1 en el cierre y continúan con la presunción de que tienen uso exclusivo del recurso. La disponibilidad de una *instrucción probar y establecer*, que lee el cierre en un registro y establece su valor en memoria compartida a 1, todo en un proceso ininterrumpible, resuelve el problema.

El segundo mecanismo útil es uno que permite *sincronización de barrera*. Suponga que se deben sincronizar p procesadores. Por ejemplo, es posible que se quiera evitar que los procesadores avancen más allá de puntos predeterminados en sus cálculos hasta que los p procesadores hayan señalado algún evento. Lo anterior se puede ver como una barrera para mayor progreso, de ahí el nombre de sincronización de barrera. El problema se soluciona al usar un contador que se inicializa en 0, incrementa para cada proceso conforme llega al punto requerido y lo prueben todos los procesos para ver si alcanzó p . Al contador se debe acceder mediante el proceso de cierre destacado en el párrafo anterior. El problema con este enfoque es que requiere muchos accesos al mismo cierre y variables de contador. Cada uno de estos accesos es un acceso a escribir con cambios de estado asociados e invalidación en las memorias caché. En virtud de que la sincronización de barrera para p procesadores constituya una función AND lógica sobre p bits, es bastante simple proporcionar un mecanismo de hardware especial para realizarla a mayor rendimiento del que es posible con el método basado en software.

■ 27.4 Memoria compartida distribuida

Los problemas de escalabilidad de los multiprocesadores con memoria centralizada da lugar a una pregunta natural: ¿la compartición de un espacio de dirección global requiere que la memoria esté centralizada? La respuesta es no. La memoria puede estar distribuida con una porción del espacio de dirección empaquetado con cada procesador. Cuando un procesador genera una dirección para acceso a memoria y la dirección se refiere a una localidad de memoria que no es local, se inicia un acceso remoto y, después de cierta latencia que es bastante grande en comparación con la latencia del acceso a memoria local, los datos solicitados regresan al procesador. El multiprocesador resultante se conoce como NUMA (acceso a memoria no uniforme), como se observó en la sección 27.1. La penalización de tiempo para acceso remoto a memoria se puede reducir mediante una combinación de operación de caché y gestión de datos (mover datos a la localidad física donde se necesitan con más frecuencia).

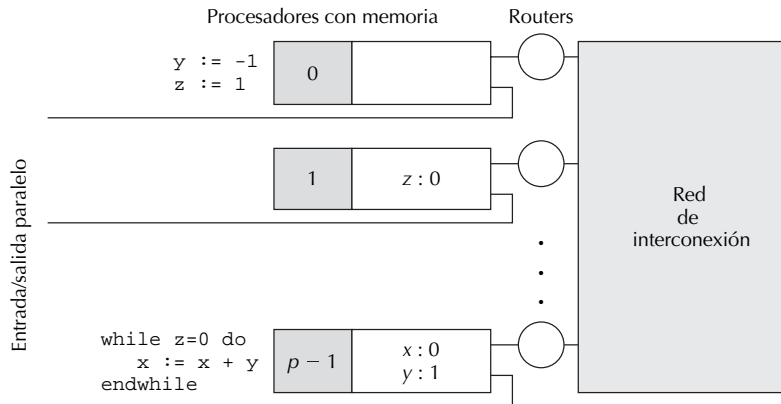


Figura 27.8 Estructura de un multiprocesador con memoria compartida.

La figura 27.8 muestra la estructura general de un multiprocesador con memoria compartida distribuida. La red de interconexión de nuevo puede variar desde un bus compartido hasta una diversidad de redes directas e indirectas (sección 28.2). La razón de que incluso un bus compartido pueda no limitar severamente la escalabilidad es que la transferencia de datos a través del mecanismo de interconexión sólo se necesita cuando se inicia un acceso a memoria no local. Con la distribución de datos adecuada, y aplicaciones típicas que tienen que ver más con la localidad de acceso, el volumen del tráfico de datos a través de la red de interconexión de la figura 27.8 puede representar una fracción muy pequeña de la de la figura 27.1 y menor que la de la figura 27.4. Otra ventaja de la memoria distribuida sobre la memoria centralizada es que el I/O paralelo de ancho de banda alto se simplifica significativamente porque las transferencias pueden ocurrir hacia/desde los módulos de memoria separados con el uso de controladores DMA convencionales sin que los datos tengan que viajar a través de las redes de interconexión.

Con el propósito de ver algunos de los problemas que puede producir el acceso a memoria no uniforme, considere los tres objetos de datos x , y y z , con las localidad y valores iniciales que se muestran en la figura 27.8, junto con fragmentos de programa simples ejecutados en dos de los procesadores. Suponga que los procesadores comienzan a ejecutar los dos fragmentos de programa al mismo tiempo. El valor final para x es una función del ordenamiento relativo de los accesos a memoria, que, a su vez, dependen de las latencias entre varios nodos en la red de interconexión. Por ejemplo, es posible que el ciclo *while* (mientras) se ejecute muchas veces antes de que el valor de y cambie a -1, y muchas más veces antes de que z cambie a 1. Por otra parte, el procesador 0 puede fijar z a 1 antes de que el procesador $p-1$ complete el acceso al valor de z , ello conduce a no ejecución absoluta del bucle.

Observe que los dos fragmentos de programa en el ejemplo anterior son partes de un programa paralelo que se ejecuta en el multiprocesador de memoria distribuida de la figura 27.8. La función pretendida de este programa paralelo no se puede deducir a partir de las pequeñas partes mostradas. Sin embargo, se puede preguntar legítimamente si alguna vez será útil un programa cuyos valores calculados dependen de ordenamiento indeterminista de eventos en hardware. Ese indeterminismo constituye una fuente clave de dificultad al diseñar programas paralelos eficientes. En el ejemplo, todo lo que el programador puede esperar razonablemente es que $y := -1$ se completará antes que $z := 1$, que las iteraciones *del ciclo while* se ejecutarán en orden natural y que, en cada iteración, el valor de z se probará antes de modificar x . En ausencia de sincronización explícita entre eventos en diferentes procesadores (por ejemplo, el ordenamiento de la asignación $z := 1$ frente a la prueba $z = 0$ no se puede predecir). En otras palabras, las instrucciones provenientes de diferentes procesadores

se pueden interpolar en cualquier forma, en tanto que las instrucciones para cualquier procesador particular aparecen en su orden especificado. Los siguientes son algunos de los posibles ordenamientos de eventos para el ejemplo que se muestra en la figura 27.8:

asignar y, asignar z, probar z, asignar x, probar z, asignar x,...
 probar z, asignar y, asignar x, probar z, asignar y, asignar x,...
 probar z, asignar x, probar z, asignar y, asignar x, asignar z,...

Una consecuencia de la expectativa por mantener el ordenamiento de instrucción en cada procesador es que, si algún procesador “ve” que z supuso el valor 1, también debe estar atento al nuevo valor -1 de y . Esta propiedad se conoce como *consistencia secuencial*, y es muy intuitiva y razonable de esperar. Con referencia al ejemplo de la figura 27.8, ningún procesador debe deducir alguna vez que se satisficiera la condición $y = z$, porque esto sería inconsistente con $y := -1$ que precede a $z := 1$. Sin embargo, asegurar consistencia secuencial no es fácil de lograr y viene con penalización de rendimiento. Observe que, para consistencia secuencial, resulta insuficiente asegurar que el cambio en el valor de y precede al cambio de z , porque un procesador que tenga latencias muy diferentes para acceder a las unidades de memoria donde se almacenan y y z todavía puede “verlos” en orden inverso. Esto es similar al hecho de que un astrónomo que ve eventos en estrellas distantes en orden cronológico inverso debido a diferentes distancias y, por tanto, tiempos de viaje de la luz que llega de cada estrella.

Observe que el problema de la consistencia secuencial no es único de los multiprocesadores de memoria distribuida y también se debe abordar en muchos sistemas con memoria centralizada; sin embargo, aquí se discutió porque tiene efectos más serios, y es más difícil de forzar, cuando la memoria está distribuida. Con el interés de lograr un mejor rendimiento, se pueden definir modelos de consistencia más relajados; pero entonces los programas paralelos se deben escribir de modo que tengan sentido, y produzcan resultados correctos, con el modelo particular supuesto. Una discusión de los diversos modelos de consistencia de memoria y sus propiedades y conflictos no es propósito de este libro.

■ 27.5 Directorios para guía de acceso a datos

La distribución estática de datos en los diversos módulos de memoria de la figura 27.8 se puede adecuar para algunas aplicaciones. No obstante, en la mayoría de los casos, aunque todavía exista localidad de acceso, el foco de atención dentro del espacio de dirección cambia dinámica para cada procesador. Esto último requiere que los elementos de datos se muevan físicamente entre los módulos de memoria, o que las copias locales (en caché) se pongan a disposición de otras localidades para maximizar la fracción de accesos que son locales. Desde luego, una reubicación de datos implica cierto desperdicio en el tiempo de transferencia, así como cabecera de gestión y no se debe usar indiscriminadamente. Se incurre en cabecera de gestión porque, al abandonar la distribución de datos estática, se debe seguir la huella de por dónde se fueron los diversos elementos de datos, con el propósito de encontrarlos cuando se necesiten. Operar en caché también involucra una cabecera para asegurar coherencia de múltiples copias. En cualquier caso, se requiere un directorio para indicar dónde se pueden encontrar objetos de datos particulares, o dónde se ubican las copias de caché (para el caso de que la invalidación se vuelva necesaria).

La discusión de esta sección continuará suponiendo que cada línea de datos tiene una localidad base asignada estáticamente dentro de uno de los módulos de memoria, pero que puede tener múltiples copias en caché en varias localidades. La figura 27.9 muestra una organización de sistema que permite este tipo de compartición de datos. Suponga que una línea caché se toma como la unidad de transferencia de datos. Asociada con cada uno de tales elementos de datos, existe una entrada de directorio que especifica cuáles cachés tienen cargada una copia de los datos. El conjunto de cachés que contienen

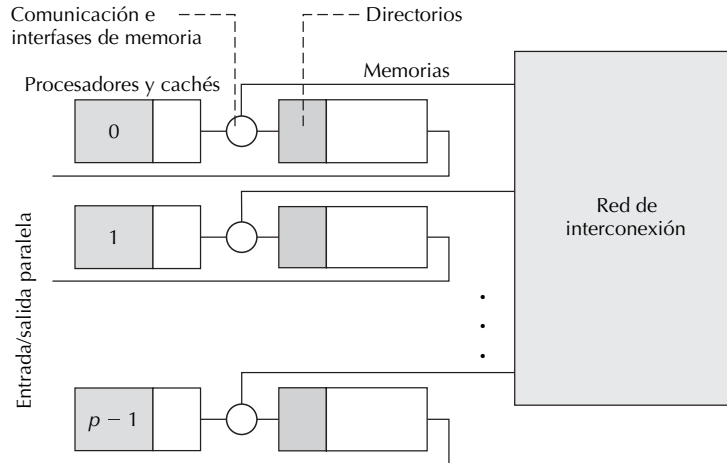


Figura 27.9 Multiprocesador con memoria compartida distribuida con una caché, directorio y módulo de memoria asociados con cada procesador.

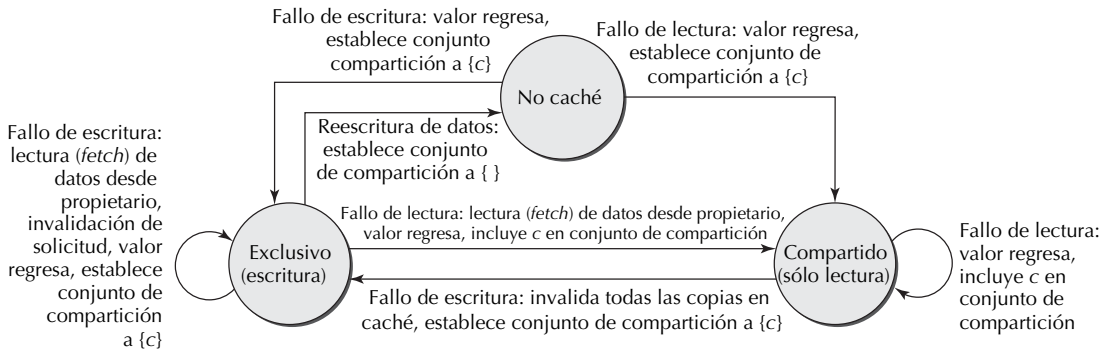


Figura 27.10 Estados y transiciones para una entrada de directorio en un protocolo de coherencia de caché basado en directorio (c es la caché solicitante).

copias representa el *conjunto de compartición* para la línea. El conjunto de compartición se puede especificar en diferentes formas. Si el número de procesadores es muy pequeño, entonces un vector de bit es bastante eficiente. Por ejemplo, con 32 procesadores se puede usar una palabra de 32 bits para mostrar cuál de las 32 memorias caché contiene una copia. Una alternativa consiste en proporcionar un listado de las cachés; esto es muy eficiente cuando los conjuntos de compartición usualmente son muy pequeños. Ninguno de estos dos métodos es escalable. Un método escalable, que se usa en el estándar *Interfaz coherente escalable* (SCI, por sus siglas en inglés), es, para la entrada de directorio, especificar una de las cachés que contienen una copia, y que cada copia contenga un puntero hacia la siguiente copia de caché. En efecto, éste es un esquema de directorio distribuido.

Como en el protocolo basado en *snoop* discutido en la sección 27.2, cada línea de caché está asociada con un mecanismo de control de estado finito que es similar al de la figura 27.5, excepto que la interacción con otras cachés se logra a través de mensajes que se envían a, y reciben desde, el directorio, en lugar de darle seguimiento a un bus. Otro mecanismo de control de estado finito se asocia con cada entrada de directorio para una sección con tamaño de línea del espacio de dirección compartido. El directorio recibe mensajes desde varias unidades caché en relación con líneas de datos particulares, actualiza su estado en el directorio y (si se requiere) envía mensajes hacia otras cachés que contengan copias de la línea de datos. La figura 27.10 muestra los estados y transiciones asociadas para una en-

trada de directorio. Los estados compartido y exclusivo son similares a sus contrapartes de la figura 27.5. Una entrada de directorio que esté en el estado no caché implica que la línea de datos no existe en alguna de las cachés. Cuando una entrada de directorio está en el estado exclusivo, la correspondiente línea de datos existe en una sola caché y se dice que está apropiada por ella. Cuando otra caché requiere una copia de dicha línea, la propietaria o envía una copia de la línea actualizada (fallo de lectura en la caché solicitante), y el estado se vuelve compartido, o reescribe la línea e invalida su copia (fallo de escritura en la caché solicitante).

Los eventos de activación para las transiciones de estado en la figura 27.10 son mensajes enviados por las memorias caché al directorio que contiene la línea solicitada. Cuando la caché c indica un fallo de lectura, ocurre uno de los siguientes eventos, después de lo cual la línea de datos se vuelve (permanece) compartida:

Línea es compartida: el valor de datos se envía a c y c se suma al conjunto de compartición.

Línea es exclusiva: se envía a la propietaria un mensaje leer (*fetch*), los datos regresados se envían a c y c se suma al conjunto de compartición.

Línea es no caché: el valor de datos se envía a c y el conjunto de compartición se fija a $\{c\}$.

Si el mensaje de caché indica un fallo de escritura, ocurre uno de los siguientes eventos, después de lo cual la línea de datos se vuelve (permanece) exclusiva:

Línea es compartida: se envía un mensaje de invalidación a las cachés que pertenecen al conjunto de compartición, el conjunto de compartición se fija a $\{c\}$ y el valor de datos se envía a c .

Línea es exclusiva: se envía a la propietaria un mensaje leer (*fetch*)/invalidar, los datos regresados se envían a c y el conjunto de compartición se fija a $\{c\}$.

Línea es no caché: el valor de datos se envía a c y el conjunto de compartición se fija a $\{c\}$.

Si de una caché se recibe un mensaje de reescritura (lo que ocurre cuando la caché necesita sustituir una línea sucia con otra línea), el estado de la línea cambia a no caché y el conjunto de compartición se vuelve vacío.

En una variación extrema de la memoria compartida distribuida conocida como *arquitectura de memoria de sólo caché* (COMA, por sus siglas en inglés), las líneas de datos no tienen bases fijas. En vez de ello, toda la memoria asociada con cada procesador está organizado como una caché, y cada línea tiene una o más copias temporales en estas memorias parecidas a caché, que se conocen como *memorias de atracción* porque atraen las líneas de datos a las que acceden sus procesadores asociados.

Un enfoque a la compartición y coherencia de datos que requiere poco o nulo soporte de hardware es la *memoria virtual compartida*. Puesto que el soporte de memoria virtual ya se proporciona en muchos sistemas, la construcción de un esquema de coherencia sobre éstos es directo. Cuando ocurre una falla de página en un nodo particular, el manipulador de falla de página obtiene la página desde una memoria remota, si se requiere, con el uso del paso de mensajes estándar. Sin embargo, en virtud de que tanto el CPU y el sistema operativo se involucran en este proceso y se debe transferir gran cantidad de datos, la cabecera de tiempo es muy grande.

■ 27.6 Implementación de multiprocesadores asimétricos

Aun cuando es posible usar un bus compartido como el mecanismo de interconexión en el multiprocesador de la figura 27.8, los beneficios de escalabilidad de la memoria compartida distribuida

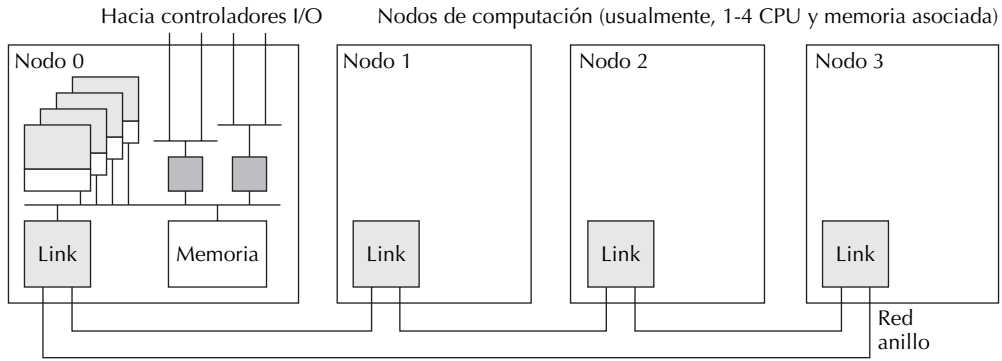


Figura 27.11 Estructura de un multiprocesador con memoria distribuida basada en anillo.

serán un poco acallados con un bus. Los multiprocesadores con memoria compartida distribuida se han implementado mediante una diversidad de redes de interconexión, que incluyen anillo, rejilla e hipercubo (sección 28.2). En esta sección se toma la más simple de estas opciones, la red anillo, para proporcionar una imagen razonablemente completa de los conflictos y negociaciones de implementación. Para reducir presentación a una longitud adecuada para este libro, se harán algunas suposiciones simplificadoras. El capítulo 8 de [Cull99] contiene una discusión mucho más extensa de los conflictos de implementación para los procesadores basados en directorio, incluidos los efectos de muchas decisiones de diseño que no se tratan aquí.

La figura 27.11 muestra la estructura de un multiprocesador con memoria distribuida basada en anillo. El sistema está estructurado en torno a una red anillo que conecta algunos nodos de computación. Cada nodo contiene uno o más procesadores con memoria caché, una memoria principal local y puentes I/O conectados a controladores I/O a través de buses I/O, todos ligados mediante un bus de nodo. El componente clave representa el recuadro denominado *Link*, que permite que un nodo de computación se comunique con otros nodos a través de la red anillo y también implementa el protocolo de coherencia entre nodos. Contiene una memoria caché que contiene datos de los otros nodos (caché remota), un directorio local y controladores de interfaz para el lado bus y el lado red. Cuando se necesitan datos remotos y no se encuentran en la caché remota dentro del recuadro *link*, se envía una solicitud a través de la red anillo. La coherencia se cumple en dos niveles. Dentro de cada nodo de computación, se usa *snooping* para asegurar coherencia entre cachés de procesador, la caché remota incrustada en el recuadro *link* y la memoria principal. La coherencia internodo se fuerza mediante un protocolo basado en directorio, que se describe dentro de poco.

El multiprocesador que se muestra en la figura 27.11 es ejemplo de arquitectura jerárquica en dos niveles que es muy popular porque reduce la complejidad estructural en cada nivel y permite que las comunicaciones locales dentro de nodos (o *clusters*) se realizan con mayor rapidez. La llegada de multiprocesadores de chip, que acomodan algunos procesadores, memorias caché y acaso controladores e interfases en un solo chip, hacen todavía más deseables tales arquitecturas multinivel.

El protocolo de coherencia que se muestra en parte de la figura 27.10, y se describe en la sección 27.5, se puede usar con cualquier representación del conjunto de compartición. La figura 27.12 muestra un método de lista ligada para representar el conjunto de compartición que es escalable a un número arbitrario de procesadores. Este método se usa en el estándar SCI. En lo que sigue, se describe el protocolo de coherencia SCI suponiendo nodos de un solo procesador. La interacción de este protocolo con el protocolo basado en *snoop* usado para forzar coherencia dentro de cada nodo se describirá más ade-

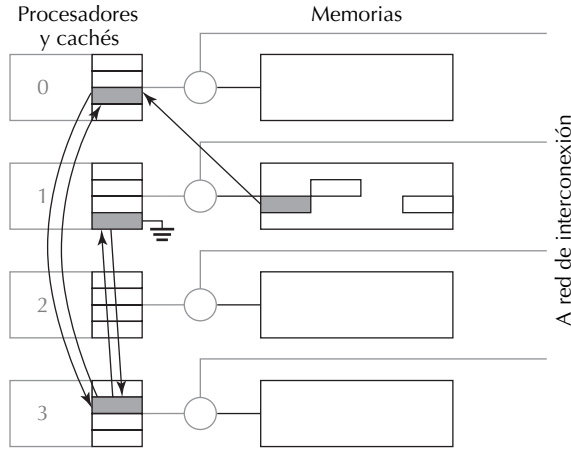


Figura 27.12 Estructura del conjunto de compartición en la interfaz coherente escalable. Los bloques sombreados son coherentes y constituyen el conjunto de compartición {0,3,1}.

lante. En SCI, las líneas de datos en memoria son de dos tipos: coherente e incoherente. La coherencia no se fuerza para el último tipo de datos con el propósito de evitar pagar la cabecera de forzamiento para los datos que no se pretende compartir o se comparten en modo de sólo lectura. La entrada de directorio y la línea de datos asociada residen en el módulo de memoria de un procesador particular. La entrada de directorio apunta a una de las cachés que contiene una copia de la línea de datos. Si la línea es no caché, entonces aparece una liga nula en el directorio. Cada caché, a su vez, contiene un puntero hacia la siguiente caché y otra a la caché previa dentro de una cadena de doble ligado. La última caché en esta cadena contiene una liga de adelantamiento nulo.

Cuando se tiene que agregar una caché al conjunto de compartición, se incluye en la lista ligada entre la entrada de directorio y la caché a la cual apunta. Por tanto, sólo necesita ocurrir un puñado de modificaciones al apuntador para este cambio en el conjunto de compartición. Cuando éste es único, como para el caso de una línea de datos exclusiva, los datos están fácilmente disponibles, cuando se necesitan, desde el caché al que apunta la entrada de directorio. Cuando una caché sobrescribe una línea y se debe remover del conjunto de compartición, simplemente notifica a las dos cachés en las direcciones adelante y atrás de modo que apunten una a otra en vez de hacia la caché removida. Sólo la caché al comienzo de la lista ligada puede invalidar líneas de datos en todas las cachés miembro del conjunto de compartición; esto último se hace al transmitir un mensaje a lo largo de la cadena hasta que alcance a la última caché. Por esta razón, la latencia de invalidación representa una función del tamaño del sistema. Por ende, aun cuando este protocolo sea escalable desde el punto de vista de representar el conjunto de compartición, no es completamente escalable desde el enfoque de rendimiento. Si una caché que quiere ganar estatus exclusivo para una línea de datos no está al comienzo de la lista, se remueve a sí misma del conjunto de compartición, luego vuelve a ganar membresía, esta vez como el elemento que encabeza la lista y finalmente procede a enviar la solicitud de invalidación.

Observe que, en la organización de la figura 27.11, cada nodo de computación representa un multiprocesador simétrico que usa un protocolo de coherencia de caché *snoopy* basado en bus. Este protocolo fuerza la coherencia dentro de las cachés de procesador, memoria principal y caché remota del nodo incrustadas en la caja *link*. A continuación se discuten las interacciones entre este protocolo de coherencia a nivel nodo y el protocolo de coherencia internodo de los párrafos precedentes. Considere primero un fallo de lectura en la caché de un procesador que se señala en el bus de nodo. En la caja de *link* averigua acerca de este fallo mediante *snooping* el bus. Verifica la caché remota, así como el directorio para líneas asignadas localmente, para ver si la solicitud se puede satisfacer dentro del nodo. De otro modo, el

recuadro *link* invoca el protocolo de directorio para obtener la línea requerida a partir de otro nodo. Si el bus de nodo usa un protocolo de transacción dividida, este evento corresponde a una respuesta tardada y no necesita manipulación especial. Mientras tanto, otros accesos dentro del nodo pueden proceder a usar el bus sin degradación de rendimiento. Un fallo de escritura se maneja de manera similar. Escribir a bloques locales es directo. Si se necesita invalidación en otros nodos, esto se hace mediante en la caja del *link* antes de que en realidad tenga lugar la escritura, y se emite un reconocimiento. Para líneas no locales, el protocolo de directorio asegura la manipulación correcta de la solicitud de escritura.

Al igual que su contraparte con memoria centralizada, un multiprocesador con memoria distribuida también puede proporcionar mecanismos para exclusión mutua y sincronización. Considere, por ejemplo, el problema de proporcionar una *instrucción probar y establecer* atómica, discutida cerca del final de la sección 27.3. Puesto que la ejecución de esta instrucción implica una operación escritura, el procesador debe adquirir una copia exclusiva de la línea que contiene la variable cierre. Luego, la instrucción se ejecuta en forma atómica en el nodo de multiprocesador simétrico, como se discutió en la sección 27.3. Otro procesador que realice probar y establecer debe adquirir su copia exclusiva después de causar invalidación para la copia exclusiva existente. Estas invalidaciones repetidas inducen mayor cantidad de cabecera. Sería más eficiente declarar tales variables de control compartidas como permanentemente no caché. Probar y establecer se envía como transacción sobre la red hacia la localidad base de la variable del cierre, se establece el cierre y su valor previo regresa al procesador que inició. Observe que, aun cuando múltiples transacciones probar y establecer se puedan traslapar durante la transmisión sobre la red, se serializan mediante el módulo de memoria que contiene la variable cierre, y a cada iniciador se envía de vuelta una respuesta correcta correspondiente a dicha serialización.

PROBLEMAS

27.1 Efectividad en costo del multiprocesamiento

El costo de un multiprocesador de p procesadores con memoria compartida, dentro de su límite de escalabilidad en términos del número de procesadores, se expresa en la forma $a + bp$, donde a representa el costo base fijo y b el costo por procesador. El costo de un uniprocador con potencia comparable a uno de los procesadores del multiprocesador es c , donde $c > b$. ¿Qué aceleración de computación se debe lograr con el sistema de p procesadores para que se le considere más efectivo en costo que la solución de uniprocador? Discuta.

27.2 Redes mariposa y de Beneš

Considere las redes de interconexión mariposa y de Beneš que se muestran en la figura 27.2.

- Presente un intercambio que no se pueda enrutar por la red de ocho hileras (16 entradas) de la figura 27.2a.
- ¿La red de la figura 27.2a se convierte en una red de intercambio si se le usa para conectar ocho procesa-

dores a ocho módulos de memoria (es decir, si los *switches* en las columnas de los extremos izquierdo y derecho son 1×2 y 2×1 , respectivamente)?

- Vuelva a dibujar la red de Beneš de la figura 27.2b, de modo que los módulos de memoria aparezcan a la derecha. Esto es, dibuje la mitad inferior de la red a la derecha de la mitad inferior, con el orden de columna invertido.
- Elimine las redundancias de la red de la parte c) al observar que las tres columnas de *switches* de en medio no conducen a un cambio en el número de hileras para los mensajes transmitidos. La red simplificada resultante es una verdadera red de Beneš.

27.3 Redes procesador-a-memoria

Considere la red de interconexión que se muestra en la figura 27.3.

- Explique por qué los bancos de memoria están numerados como se muestra en el lado derecho de la figura. *Sugerencia:* El paso de 1 es el paso más común usado en accesos a memoria.

- b) ¿Qué paso de acceso causaría más dificultades con esta red?
- c) ¿La red representa una red de intercambio? ¿Por qué?

27.4 Máquina de acceso aleatorio paralelo

Considere los dos algoritmos PRAM para transmitir y sumar los elementos vectoriales presentados al final de la sección 27.1.

- a) Reescriba el algoritmo de transmisión de modo que ningún procesador intente leer o escribir elementos que no pertenezcan al vector B .
- b) Aplique el método de duplicación del algoritmo de suma al algoritmo de transmisión de modo que se evite calcular una potencia de 2 en cada ejecución del ciclo.
- c) Combine los algoritmos de las partes a) y b) en un solo algoritmo que llene todos los elementos del vector Z resultante con la suma calculada.
- d) Suponga que se tiene interés en calcular todas las sumas parciales $Z[i] := X[0] + X[1] + \dots + X[i]$ en lugar de sólo la suma global $X[0] + X[1] + \dots + X[p-1]$. Modifique el algoritmo de suma para producir todas las sumas parciales requeridas.

27.5 Coherencia de valores de registro

Estrictamente hablando, los registros forman parte de la jerarquía de memoria. En los registros se cargan valores de variable, se les modifica a través de uno o más pasos de cálculo y se les almacena de vuelta. ¿Por qué la coherencia no es un problema con los registros, como lo es con la memoria caché?

27.6 Protocolos de coherencia basados en *snoop*

Considere el protocolo de coherencia caché basado en *snoop* que se muestra en la figura 27.5.

- a) ¿Hay algún beneficio en agregar un cuarto estado “privado” (no compartible) a la máquina de control de estado finito del protocolo? Justifique su respuesta.
- b) Modifique la máquina de control de estado finito para que corresponda a un sistema multiprocesador con cachés a través de escritura. Discuta los cambios en sus impactos costo/rendimiento.

27.7 Implementación de un protocolo basado en *snoop*

Considere el *buffer* de escritura que contiene una línea caché a escribir a memoria principal en la figura 27.7. ¿Por qué la tag de *buffer* de escritura está conectada al de dirección para acceso a memoria abajo a la derecha del diagrama?

27.8 Límite de escalabilidad con cachés a través de escritura

Un multiprocesador basado en bus tiene escalabilidad limitada para comenzar, pero ésta se puede volver incluso más limitada si se usan cachés a través de escritura. Considere el uso de procesadores de 1 GHz con un CPI promedio de 0.8 para construir un multiprocesador basado en bus con memoria compartida centralizada. Suponga que 10% de todas las instrucciones se almacenan con ancho de datos de ocho bytes.

- a) ¿Cuál es una cota superior en el número de procesadores que puede soportar un bus de 4 GB/s?
- b) ¿Cuál es la cota inferior sobre el ancho de bus necesario para soportar 32 procesadores?

27.9 Ancho de línea óptimo para cachés

En la sección 18.6 se discuten los beneficios e inconvenientes de líneas caché más anchas y se afirmó que con frecuencia se puede determinar un ancho de línea caché óptimo. Discuta si en un multiprocesador con memoria compartida con base en bus, como el de la figura 27.6, es probable que una línea caché más estrecha o más ancha pueda ser óptima en comparación con la de un uniprocesador. *Sugerencia:* Es posible que los dos procesadores invaliden repetidamente los datos mutuos incluso cuando no comparten alguna variable. Esto se denomina *compartición falsa*.

27.10 Coherencia de caché con actualización

Algunos protocolos de coherencia de caché actualizan (en lugar de invalidar) las otras copias de una línea caché cuando se modifica una copia compartida. La ventaja obvia de actualizar es que evita algunos fallos de caché futuros. El inconveniente es que la actualización produce tráfico de bus o red adicional. Discuta qué enfoque, invalidación o actualización, es probable que conduzca

a mejor rendimiento con cada uno de los siguientes patrones de acceso a una sola variable compartida. Establezca claramente todas sus suposiciones.

- a) Repita muchas veces: un procesador escribe un nuevo valor en la variable compartida y los otros $p - 1$ procesadores leen el nuevo valor.
- b) Repita muchas veces: un procesador escribe m veces en una variable compartida; esto es seguido por otro procesador que lee el valor de la variable compartida.

27.11 Consistencia secuencial

Una máquina paralela de memoria compartida presenta la ilusión de una sola unidad de memoria para todos sus procesadores. Se dice que tal multiprocesador proporciona *consistencia secuencial* (también conocida como *consistencia fuerte*) si su sistema de memoria se comporta como si hubiese una sola unidad de memoria que diferentes procesadores toman turnos para usar. Por ende, los datos escritos por un procesador en una localidad de memoria particular quedan inmediatamente “visibles” a todos los procesadores. El efecto de este comportamiento en la parte del sistema de memoria es que los procesadores producen resultados que se habrían producido si los accesos a cada localidad de memoria se hubiesen serializado en algún orden.

- a) Explique por qué la consistencia secuencial puede no ser proporcionada naturalmente en un multiprocesador de memoria distribuida y necesita esfuerzo adicional para cumplirse.
- b) Dado que el cumplimiento de la consistencia secuencial conlleva penalizaciones de rendimiento, con frecuencia se usan los modelos de *consistencia relajada* con la intención de permitir mayor concurrencia en accesos a memoria [Adve96]. Identifique dos modelos de consistencia relajada y compárelos uno con otro y con consistencia secuencial para facilidad de programación, costo de implementación e impacto de rendimiento.

27.12 Memoria remota en una jerarquía de memoria

Las lecturas de latencias de acceso para un multiprocesador de memoria compartida distribuida son las siguientes: cinco ciclos para un fallo de caché primaria que impacta en la caché secundaria en chip; 15 ciclos para un fallo de caché secundaria que impacta en la ca-

ché terciaria (nivel de tarjeta); 50 ciclos para un fallo que es atendido por la memoria; 100 ciclos para un fallo que tiene atenderse a partir de la memoria de otro procesador. Determine el tiempo de acceso de lectura promedio en términos de las tasas de impacto en los varios cachés y en memoria local.

27.13 Entradas de directorio para varios datos

Considere la figura 27.4, que muestra varios tipos de datos en caché. Agregue anotaciones similares a la de la figura 27.9 y, en cada caso, defina la correspondiente entrada de directorio.

27.14 Coherencia de caché basada en directorio

La figura 27.10 muestra el mecanismo de control de estado finito asociado con cada entrada de directorio en un protocolo de coherencia de caché basado en directorio. Proporcione el control de estado finito que se debe asociar con las líneas de caché. *Sugerencia:* El diagrama requerido es similar al de la figura 27.5.

27.15 Protocolos de coherencia basados en directorio

La discusión de los protocolos de coherencia caché basada en directorio de la sección 27.5 se basó en *directorios planos*, llamados así porque la información de directorio para cada bloque se encuentra en una localidad fija determinada únicamente por su dirección. Cuando ocurre un fallo, una sola petición se envía al nodo base único para ciclo de directorio, cuyo resultado se guía entonces al acceso de datos actual. En contraste, los directorios jerárquicos están organizados como árboles lógicos incrustados en el esquema de interconexión particular que usa el sistema. Cada bloque está asociado con un árbol lógico particular y las solicitudes a dicho bloque siempre se adelantan por un nodo a su padre dentro de dicho árbol lógico. Estudie tales directorios jerárquicos [Cull99] y escriba un breve informe que incluya la siguiente información.

- a) Costo de implementación en comparación con los directorios planos.
- b) Beneficios de rendimiento, para el caso de que existiera alguno, en relación con directorios planos.
- c) Métodos para definir y mantener los árboles lógicos.
- d) Aspectos de escalabilidad, confiabilidad y fortaleza.

27.16 Multiprocesadores simétricos reales

Seleccione un multiprocesador simétrico real y prepare un informe acerca de su diseño, dedicando al menos una página para describir cada uno de los aspectos siguientes. Sequent, SGI y Sun son algunas de las compañías que fabrican tales sistemas.

- a) Componentes de hardware básicos (procesadores, cachés, memoria).
- b) Interconexión procesador a memoria.
- c) Protocolo de coherencia caché.

27.17 Multiprocesadores asimétricos reales

Seleccione un multiprocesador asimétrico real y prepare un informe acerca de su diseño, dedicando al menos una página para describir cada uno de los aspectos siguientes. IBM, Sequent y SGI son algunas de las compañías que fabrican tales sistemas.

- a) Estructura de nodo en términos de procesadores y memorias caché.
- b) Red de interconexión.
- c) Protocolo de coherencia de caché.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Adve96] Adve, S. V. y K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial", *IEEE Computer*, vol. 29, núm. 12, pp. 66-76, diciembre de 1996.
- [Cull99] Culler, D. E. y J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1999.
- [Henn03] Hennessy, J. L. y D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 3a. ed., 2003.
- [Hord93] Hord, R. M., *Parallel Supercomputing in MIMD Architectures*, CRC Press, 1993.
- [Leno95] Lenoski, D. E. y W. D. Weber, *Scalable Shared-Memory Multiprocessing*, Morgan Kaufmann, 1995.
- [Parh99] Parhami, B., *Introduction to Parallel Processing: Algorithms and Architectures*, Plenum Press, 1999.
- [Unge02] Ungerer, T., B. Robic y J. Silc, "Multithreaded Processors", *Computer J.*, vol. 45, núm. 3, pp. 320-348, 2002.
- [WWW] Nombres Web de algunos de los fabricantes de multiprocesadores de memoria compartida:: fujitsu.com, hp.com, ibm.com, sgi.com, sun.com.

■ CAPÍTULO 28

MULTICOMPUTACIÓN DISTRIBUIDA

“Un sistema distribuido es aquel en el que la falla en una computadora que ni siquiera sabías que existía, puede dejar inservible tu propia computadora.”

Leslie Lamport, 1992

“Una persona con un reloj sabe qué hora es; una persona con dos relojes nunca está segura.”

Proverbio

TEMAS DEL CAPÍTULO

- 28.1** Comunicación mediante paso de mensajes
- 28.2** Redes de interconexión
- 28.3** Composición y enrutamiento de mensajes
- 28.4** Construcción y uso de multicomputadoras
- 28.5** Computación distribuida basada en red
- 28.6** Computación en retícula y más allá

Los arquitectos de computadoras han imaginado un día en que será posible conectar un número indeterminado de procesadores y módulos de memoria requeridos para construir una máquina con potencia computacional y capacidad de almacenamiento deseados, en gran parte como se conectan bloques de juguete para construir estructuras complejas. Conforme crecen las necesidades de almacenamiento y computación, simplemente se unirían unos cuantos bloques más para llevar las capacidades del sistema en línea con las necesidades. Aunque este anhelo todavía es irrealizable, los avances recientes en procesador, memoria y tecnologías de comunicación lo han acercado mucho en algunas formas importantes. En este capítulo se revisan las estrategias para construir multicomputadoras a partir de nodos holgadamente conectados que se comunican mediante intercambio de mensajes. Ésta es una área atractiva de investigación y desarrollo, se han intentado muchos esquemas alternativos.

■ 28.1 Comunicación mediante paso de mensajes

Las multicomputadoras distribuidas holgadamente acopladas, o *multicomputadoras* para abreviar, se forman al interconectar un conjunto de computadoras independientes mediante una red de interconexión que les permite comunicarse mediante el paso de mensajes. La figura 28.1 muestra la estruc-

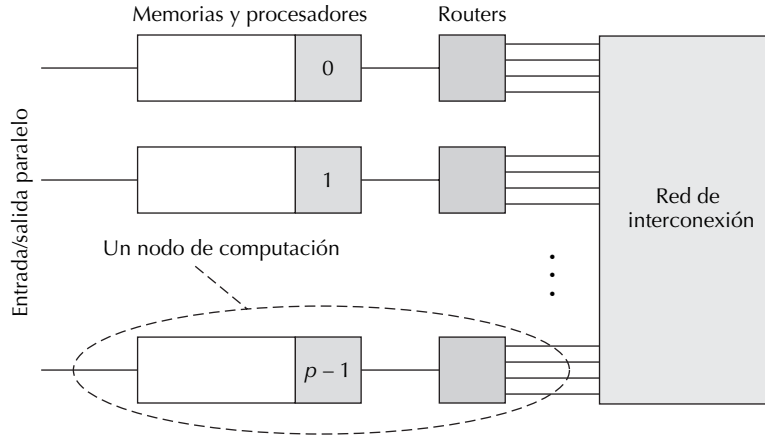


Figura 28.1 Estructura de una multicomputadora distribuida.

tura de tales multicomputadoras. A primera vista, la figura 28.1 parece similar al multiprocesador con memoria compartida distribuida de la figura 27.8. Sin embargo, una inspección más cercana revela un número de sutiles diferencias en la estructura. Esto último, a su vez, conduce a diferentes modelos de programación y dominios de aplicación. La primera diferencia clave es que la transmisión (*broadcasting*) de datos a través de la red de la figura 28.1 ocurre no como resultado de solicitudes de acceso a memoria que se refieren a memorias remotas, sino debido a la ejecución de comandos de paso de mensaje en el programa que se ejecuta. Por tanto, a diferencia de la norma de la figura 27.8, aquí está el procesador, no el controlador de memoria, que activa el ruteador (*router*) de nodo.

La red de interconexión puede constituir un simple medio de transmisión compartido (como un bus o Ethernet), una colección de alambres sin capacidad lógica o de toma de decisiones (red directa) o un sistema de ruteadores o *switches* que dirijan un mensaje desde un nodo fuente hacia sus destinos pretendidos en muchos pasos (red indirecta). La segunda diferencia clave entre las figura 28.1 y 27.8 es que, en una multicomputadora, usualmente hay más de una conexión de cada nodo con la red de interconexión. Las conexiones múltiples son necesarias para el caso de redes de interconexión directa y permite el enrutamiento a través de un pequeño número de *switches* para redes indirectas (sección 28.2). Por ahora, la red de interconexión se ve como un mecanismo que puede aceptar mensajes inyectados desde nodos fuente, los guía a través de rutas elegidas adecuadamente y los entrega al(los) nodo(s) destino pretendido(s).

Con una red de interconexión de medio compartido, la fase de enrutamiento se elimina y la comunicación degenera en adquisición de bus seguida por colocación de datos en el bus en el nodo fuente y la detección de dirección seguido por descarga de datos desde el bus al destino, todo lo anterior de acuerdo con el protocolo de medio compartido (capítulo 23).

La comunicación directa, o punto a punto, es igualmente simple porque conlleva conectar físicamente un puerto de salida del ruteador del nodo remitente hasta un puerto de entrada del receptor. Con este método, sólo un pequeño subconjunto de nodos es directamente accesible a partir de un nodo fuente (por ejemplo, 4 en la figura 28.1). Para que un mensaje llegue a otros nodos, se debe retransmitir o adelantar mediante uno o más nodos intermedios. El protocolo para realizar esto último se programa en la unidad de control del ruteador. En otras palabras, al momento de recibir un mensaje entrante, el ruteador examina su dirección de nodo de destino. Si esta dirección equipara la dirección de nodo actual, entonces el mensaje se expulsa al *buffer* de entrada del nodo local (o cola de entrada). De otro modo, se elige uno de los *links* salientes para adelantar el mensaje a un nodo diferente. La selección

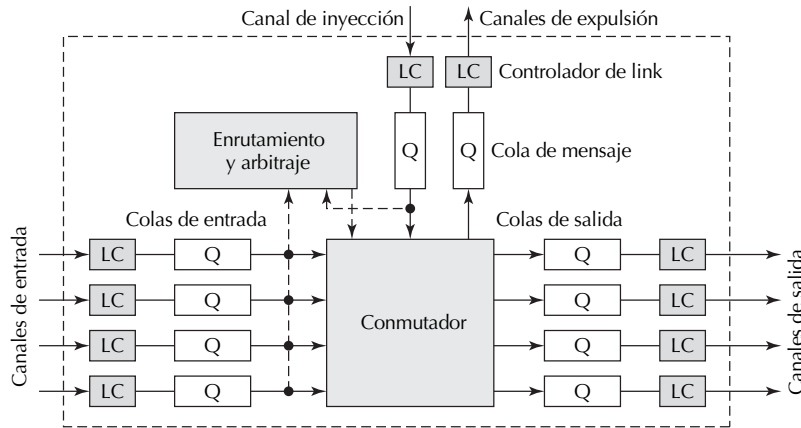


Figura 28.2 Estructura de un ruteador genérico.

es una función del algoritmo de enrutamiento que se use, el estatus de varios canales en la red y la información incrustada en el mensaje mismo. Lo último puede incluir tags que especifiquen prioridad, tamaño y latencia de mensaje ya experimentados en la red.

La figura 28.2 muestra la estructura de un ruteador genérico. El nodo local conectado al ruteador envía mensajes en la red a través del *canal de inyección* y remueve mensajes entrantes dirigidos a él a través del *canal de expulsión*. Los mensajes inyectados localmente compiten con los mensajes que pasan a través del ruteador para uso de los canales de salida disponibles, cada mensaje que no se puede adelantar se conserva en la cola asociada hasta el siguiente ciclo de arbitraje. Las decisiones de enrutamiento (elección de canal de salida) se pueden hacer en diferentes formas, incluido el uso de tablas de enrutamiento dentro del ruteador o seguridad en las etiquetas de enrutamiento llevadas con los mensajes. Las estrategias de enrutamiento incluyen *enrutamiento empaçado*, en el que todo un mensaje o una parte considerable de éste se almacena en el ruteador antes de que se adelante al siguiente nodo, y el *enrutamiento en hoyo de gusano*, donde se adelanta una porción de un mensaje (un *flit*), con la esperanza de que los *flits* restantes seguirán en rápida sucesión, en forma muy parecida a un gusano que se mueve a través de una manzana.

Además de ser puntos finales de comunicación, como en la figura 28.1, los ruteadores se pueden interconectar para formar la red de interconexión, usando estructuras similares a las redes de área local y ancha de la figura 23.2. En la práctica, es posible combinar estas dos funciones de ruteadores con el uso de un subconjunto de los canales ruteadores para inyección y expulsión, y el resto para recibir y adelantar. Este enfoque oscurece la distinción entre redes directas e indirectas al permitir muchas organizaciones intermedias.

Del mismo modo se pueden usar *switches* (conmutadores) para construir redes de interconexión. En la práctica, los *switches* pueden contener colas y lógica de enrutamiento que los hacen muy similares a los ruteadores. Sin embargo, también se pueden usar *switches* muy simples, que establecen conexiones sólo entre sus entradas y salidas, en una red de interconexión de *conmutación de circuito*. En este caso, se establece una ruta a través de la red y entonces los datos se envían sobre dicha ruta desde una fuente hacia un destino. Los *switches* de barra cruzada, con p^2 *switches* de punto de cruce entre líneas p horizontal y p vertical, ello permite conexión entre p entradas y p salidas en un patrón arbitrario. Un *switch* 2×2 simple, que es capaz de conexión punto a punto o transmisión, se muestra en la figura 28.3. En la figura 26.9 se presenta un *switch* 3×3 , con capacidad de permutación limitada, en conexión con I/O en un procesador matricial.

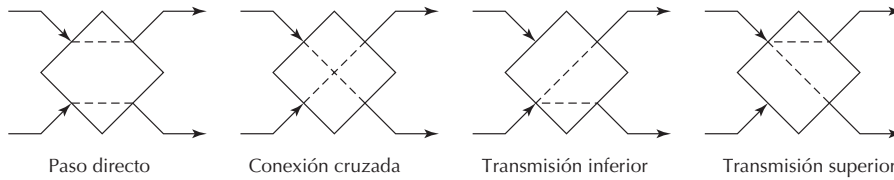


FIGURA 28.3 Ejemplo de switch 2×2 con capacidades de conexión punto a punto y de transmisión.

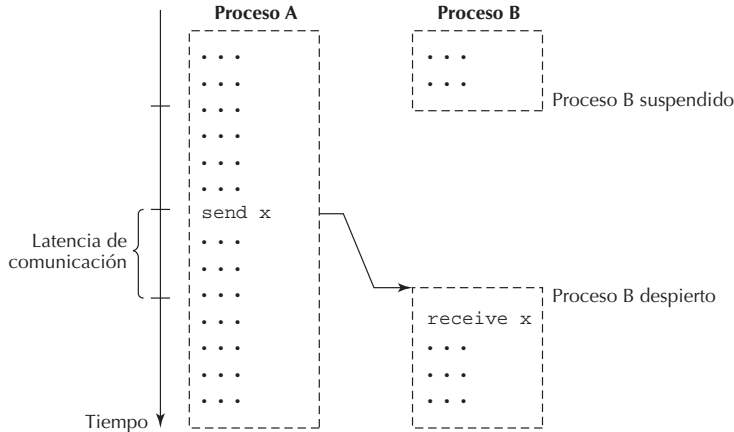


FIGURA 28.4 Uso de los primitivos mensajes de paso *send* y *receive* para sincronizar dos procesos.

Sin importar cómo los mensajes encuentran su camino desde sus fuentes hasta su destino, existen primitivos a nivel de aplicación para transmisión y recepción de mensajes que se pueden usar para construir programas paralelos. Estos primitivos proporcionan un método abstracto independiente de hardware para especificar el comportamiento deseado del programa en un entorno de pase de mensaje. Como ejemplo, considere los procesos A y B que corren en diferentes nodos de una multicomputadora, donde A requiere enviar un valor de datos a B antes de que B pueda completar su cálculo. De este modo, el proceso B es dependiente de datos del proceso A. En tal dependencia simple de una vía se puede permitir que el proceso A complete la ejecución, almacene sus resultados en memoria y luego inicie el proceso B cuyos datos requeridos ahora están disponibles. Un mejor enfoque, que permite la concurrencia entre los procesos así como entre dependencias mutuas más complicadas, es usar el paso de mensaje que se muestra en la figura 28.4.

Una forma de permitir envío-recepción y otros tipos de comunicación interprocesos es incorporar los primitivos requeridos como una librería de funciones dentro de lenguajes de programación estándar de alto nivel. El estándar *Interfaz de paso de mensaje* (MPI) es una de tales librerías que tienen la intención de proporcionar portabilidad de aplicación entre computadoras paralelas. MPI incluye funciones para comunicación punto a punto (como en enviar-recibir) así como varios tipos de comunicación colectiva:

Transmisión: un nodo que envía un mensaje a todos los nodos

Dispersión: un nodo que envía distintos mensajes a cada nodo

Recopilación: un nodo que recibe un mensaje de todos los nodos

Intercambio completo: todos los nodos realizan dispersión y recopilación

Debido a sus actividades de comunicación, como las referencias de memoria, exhibe localidad (en el sentido en que un proceso que envía un mensaje a otro es probable que lo haga de nuevo en el futuro cercano), MPI incluye características para *comunicación persistente* que permite cierta cabecera de mensajes de envío y recepción para compartirse a través de múltiples transmisiones. Adicionalmente, hay facilidades para sincronización de barrera (como se definió al final de la sección 27.3) y operaciones de reducción global. Para más detalle en MPI, consulte [Snir96].

■ 28.2 Redes de interconexión

La estructura y función de las redes de interconexión se entienden mejor a través de la analogía de rejilla de electricidad. Ésta consta de nodos (estaciones y subestaciones) y *links* (líneas de transmisión) que conectan los nodos. Algunos de estos últimos están asociados con generación de electricidad local, que es análogo a las fuentes de generación de mensaje. Otros nodos, no necesariamente distintos del conjunto previo, representan consumo de electricidad local dentro de comunidades. Las últimas son análogas a los destinos de mensaje. Finalmente, algunos nodos son simplemente relevadores o convertidores de voltaje, sin que tengan algún consumo o generación de electricidad local. Éstos son análogos a los *switches* o ruteadores. En las redes de interconexión, las fuentes de mensaje y destino representan generalmente el mismo conjunto: los nodos de computación. Cuando cada nodo en el sistema constituye un nodo de computación, la red se conoce como *red directa*, pues todos los *links* conducen directamente de un nodo de computación a otro. Cuando los nodos de computación están separados por al menos un nodo de no computación, se tiene una *red indirecta*. Como se mencionó en la sección 28.1, son posibles diseños intermedios que dificultan la diferencia entre redes directas e indirectas. La figura 28.5 muestra ejemplos de redes directas e indirectas con topologías similares.

Se ha implementado gran variedad de redes de interconexión directa. Éstas se pueden representar mediante gráficas que muestran la conectividad de sus ruteadores. Puesto que cada ruteador está asociado con un nodo de computación, los nodos mismos no necesitan mostrarse y los círculos que representan ruteadores se pueden referir como nodos. La figura 28.6 muestra algunas de tales redes, todas de tamaño $p = 16$. La red *toro 2D* se obtiene a partir de una malla 2D que conecta mutuamente los nodos en los dos extremos de una hilera o columna a través de un link de enrollado. Cada nodo se numera por un par de enteros que denotan los números de hilera y columna. Esta estructura se generaliza a variantes no cuadradas y a más de dos dimensiones en forma directa. La red *qD hipercubo* se define recursivamente del modo siguiente.

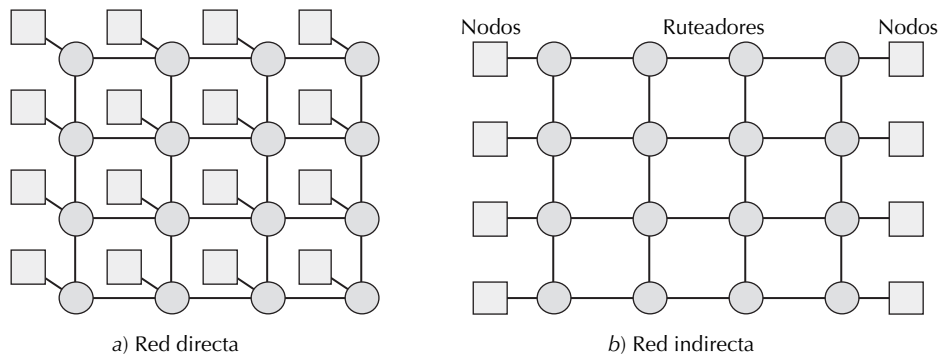


Figura 28.5 Ejemplos de redes de interconexión directa e indirecta.

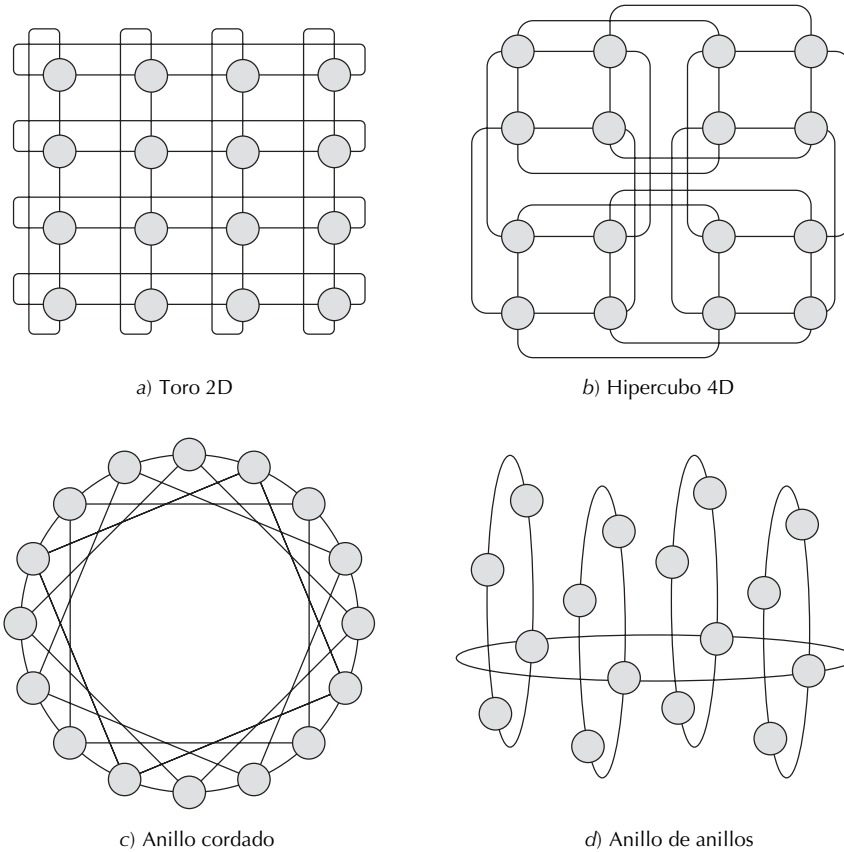


FIGURA 28.6 Muestra de redes de interconexión directa comunes. Sólo se muestran los ruteadores; para cada ruteador está implícito un nodo de cálculo.

Un hipercubo 1D es simplemente un par de nodos, etiquetados 0 y 1, que están mutuamente ligados. A partir de dos hipercubos $(q - 1)$ D se obtiene un hipercubo q D al conectar sus nodos correspondientes uno a otro con el uso de un total de 2^{q-1} nuevas ligas. Más aún, las etiquetas en uno de los $(q - 1)$ D subcubos está precedida con 0 y las del otro por 1. Un *anillo cordado* es un anillo al que se han agregado algunas conexiones de paso (*bypass*), o cuerdas. Los nodos están numerados del 0 al $p - 1$ en su orden natural en torno al anillo. En el ejemplo de la figura 28.6c, las cuerdas conectan cada nodo i con el nodo $i + 4 \pmod{16}$. Se puede proporcionar más de una cuerda por nodo, o las cuerda pueden tener diferentes distancias de *bypass*, lo que conduce a muchas variaciones. El ejemplo final de una red de interconexión directa, en la figura 28.6d, se conoce como anillo de anillos. Se trata simplemente de una colección de anillos conectados unos a otros al incluir un nodo de cada anillo dentro de un anillo de segundo nivel. La generalización a más de un anillo de segundo nivel, o a más de dos niveles de anillos, es directa.

Las redes de interconexión indirecta se pueden representar de igual modo mediante gráficas que especifiquen la conectividad de los switches o ruteadores. Sin embargo, puesto que no cada switch o ruteador está conectado a un nodo de cálculo, lo último también se debe mostrar o especificar sus ubicaciones.

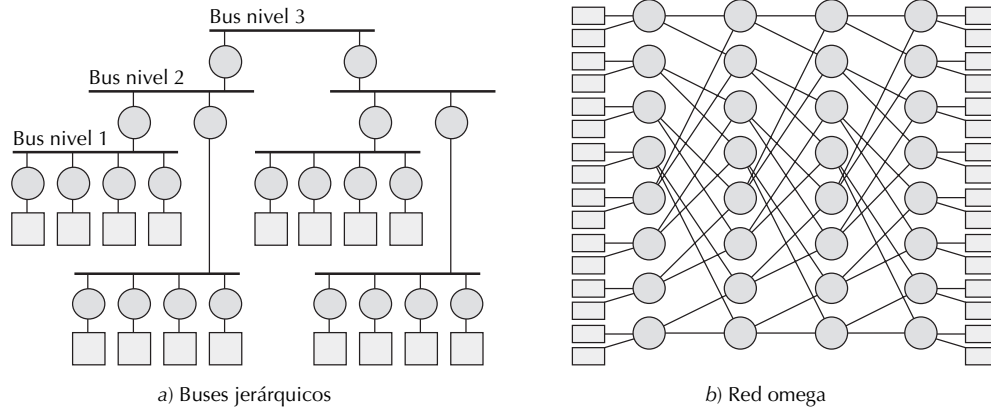


Figura 28.7 Dos redes de interconexión indirecta usados comúnmente.

La figura 28.7 muestra dos ejemplos de redes de interconexión indirecta. Las *estructuras de bus multinivel o jerárquica* son muy populares porque permiten la construcción de grandes multicomputadoras mientras superan las limitaciones de los buses en términos de escalabilidad y ancho de banda de comunicación. Las comunicaciones locales se realizan a través de buses de nivel 1, mientras que las transferencias de datos globales se realizan a través de buses de nivel superior con mayor latencia. En tanto las comunicaciones locales dominan en términos de frecuencia, resulta en un rendimiento global aceptable. Observe que, en este caso, a los ruteadores de entrada sencilla, salida sencilla con frecuencia se les denomina interfases de bus (junto a nodos de computación) o puentes de bus (en cualquier otra parte). Observe también que, estrictamente hablando, el diagrama en la figura 28.7a no representa una gráfica sino una hipergráfica porque contiene bordes que conectan más de dos nodos. Otro ejemplo de red indirecta es la *red omega* multiestado, que se muestra en la figura 28.7b. Aquí, los nodos de computación que se muestran a la izquierda y derecha de la red de conmutación usualmente constituyen el mismo conjunto de nodos (la figura se enrolla). En cada columna de esta red omega de ocho hileras, el *switch* ubicado en la hilera i se conecta con los *switches* en las hileras $2i \bmod 8$ y $2i + 1 \bmod 8$ en la siguiente columna a la derecha.

Las redes de interconexión se postulan con base en un número de propiedades que afectan su costo de implementación y rendimiento. El costo se relaciona con la complejidad de nodo, así como con la densidad y regularidad del alambrado entre nodos. Más alambres generalmente significa mayor costo. Esto sucede porque, en el nivel de chip VLSI, alambrado más denso implica mayor área de chip; más aún, a niveles superiores de la jerarquía de empaquetado, las limitaciones en pines y otros conectores fuera de módulo pueden forzar el uso de más módulos (chips, placas, etcétera) para acomodar todas las conexiones requeridas. Los alambres cortos y regularmente conectados, que se encuentran en las redes malla 2D, permiten mayor rendimiento debido a propagación de señal más rápida y controladores más simples. Los alambres largos no locales imponen el doble de penalización de la mayor complejidad y menor rendimiento.

Además del efecto indirecto del patrón de alambrado y densidad sobre la rapidez de comunicación, el rendimiento de la red de interconexión es directamente afectado por dos parámetros de red clave. El primero es el *diámetro* de red, que se define como el número de saltos (*hops*) (transferencia ruteador a ruteador) necesarios en la más larga de las rutas más cortas entre cualquier par de nodos. Por ejemplo, el diámetro de la red toro de la figura 28.6a es 4. Un diámetro más grande generalmente significa mayor latencia de peor caso en la transmisión de mensaje entre nodos. El segundo es el *ancho de bisección* de la red, que se define como el número mínimo de canales que se deben cortar si la red de

p nodos se debe dividir en dos partes que contengan $\lfloor p/2 \rfloor$ y $\lceil p/2 \rceil$. Por ejemplo, el ancho de bisección de la red toro de la figura 28.6a es 8. Mientras más grande sea el ancho de bisección de una red, mayor es el ancho de banda de comunicación disponible para transmitir mensajes de un lado a otro de la red. Esto es importante cuando el tráfico de comunicación entre nodos es aleatorio. Cuando la comunicación es predominantemente local, incluso la red de bus jerárquico de la figura 28.7a con su pequeño ancho de bisección de 1 puede ofrecer alto rendimiento. Observe que este estrecho canal que biseca puede superar un problema si gran cantidad de mensajes deben fluir a través del bus de nivel 3.

■ 28.3 Composición y enrutamiento de mensajes

Los mensajes se pueden ver en diferentes niveles de abstracción. En el nivel de aplicación, un mensaje representa una cadena de byte de longitud variable. Desde el punto de vista de los programas de aplicación que envían o reciben mensajes, se asocian significados específicos con dichas cadenas de byte (por ejemplo, una secuencia de enteros o números en punto flotante). Sin embargo, el mecanismo que implementa el paso de mensajes no necesita preocuparse por el significado de un mensaje, sólo con la entrega correcta de la cadena de bytes a su destino pretendido. Los mensajes largos o de longitud variable se pueden dividir en *paquetes* de tamaño fijo para transmisión eficiente, como se muestra en la figura 28.8. Un paquete, o todo el mensaje para el caso de paquetes que no se usan, se adiciona con un *encabezado* (*header*) y posiblemente con un *remolque* (*trailer*), que contienen información de control y verificación de error. El *header*, por ejemplo, puede contener un número de secuencia de paquete y la identidad de los nodos emisores y receptores, así como ciertos campos de control que identifican la longitud y tipo del mensaje o la ruta a tomar a través de la red. Estos componentes de un mensaje son similares a sus contrapartes usadas en redes de cómputo, excepto que, debido al número limitado de posibles fuentes y destinos, la cantidad de información de control usualmente es menor. Por ejemplo, en una computadora paralela de 256 procesadores, los identificadores de nodo fuente y destino son números de ocho bits. A su vez, un paquete se puede dividir en *dígitos de control de flujo* (*flow control digits*, *flits*, para abreviar) cuyo tamaño es de pocas a muchas decenas de bits.

La estrategia de conmutación de red determina cómo se enrutan los mensajes. La *conmutación de circuitos*, que requiere el establecimiento de un circuito físico entre la fuente y el destino antes de transmitir un mensaje, y durante toda la transmisión, se usa rara vez en las computadoras modernas porque es muy derrochadora. Con la *conmutación de paquetes*, los paquetes se enrutan por separado desde la fuente hasta el destino, donde se vuelven a ensamblar en el mensaje original con la información de secuenciación dentro de los paquetes. Esto es análogo a un gran grupo de personas que viajan a su destino común en muchos autobuses, y cada autobús toma una ruta independiente y llegan en un tiempo diferen-

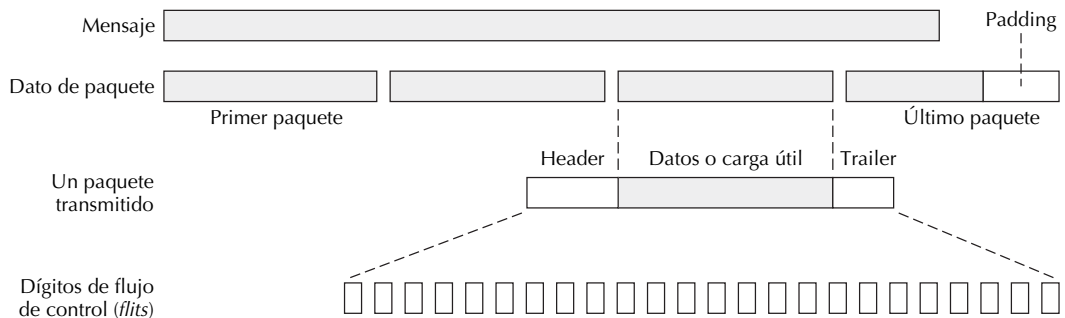


Figura 28.8 Mensajes y sus partes para paso de mensaje.

te. Si un paquete debe pasar a través de nodos intermedios o ruteadores, todo el paquete se almacena en el nodo intermedio antes de adelantarse al siguiente punto en la ruta a su destino. Por esta razón, también se usa el término “conmutación *almacenar y adelantar*”. La elección del tamaño de paquete constituye una importante decisión de diseño. Los paquetes más pequeños tienden a usar el ancho de banda de la red de manera más eficiente, pero implican mayor cabecera para los *headers* y *trailers* de paquete.

La *conmutación en hoyo de gusano* representa otra opción que se ha usado ampliamente porque combina las mejores características de la conmutación de circuito y la de almacenar y adelantar. Para continuar con la analogía del autobús, un grupo grande de personas que se dirigen a un destino común pueden abordar un tranvía con muchos carros en lugar de algunos autobuses independientes. El carro líder del tranvía dirige su ruta a través de las calles de la ciudad, elige una ruta abierta y evita las calles congestionadas para reducir el tiempo de viaje. Los carros restantes sólo siguen al líder. Si éste se detiene ante la luz roja de un semáforo que indica “alto”, los otros también se detienen. Si otros viajeros cruzan rutas con el tranvía, es posible que tengan que esperar hasta que este último pase. Este tipo de bloqueo representa un atributo negativo que necesita análisis y manejo cuidadosos. Sin embargo, en recompensa se obtiene cabecera más baja, tamaño de *buffer* más pequeño (cada nodo intermedio necesita almacenar sólo un *flit*) y retardos de transmisión más cortos. La denominación “conmutación hoyo de gusano” refleja la similitud de movimiento de datos en la red con este método, con el movimiento de un gusano adentro de una manzana. A un mensaje que se dispersa a través de múltiples nodos conforme viaja a través de la red se le refiere como *gusano* (no debe confundirse con los gusanos de Internet). La figura 28.9 muestra dos gusanos conforme se mueven a través de la red y cómo se pueden bloquear los gusanos mutuamente, ello causa bloqueos (*deadlocks*). Observe que la analogía del tranvía es imperfecta en el sentido de que los gusanos en realidad pueden cruzarse mutuamente si entran y salen de un nodo a través de diferentes *links* pero no pueden viajar a lo largo de éste en la misma dirección (como en las calles de un solo sentido).

Con cualquier conmutación de paquete o de hoyo de gusano, se requiere un procedimiento para decidir cuál de las muchas rutas desde el nodo fuente hacia el nodo destino tomará el paquete o gusano. Incluso cuando la ruta es única, la información de fuente y destino de algún modo deben representar la elección de un *link* saliente en cada paso a lo largo del camino. Esto es función del *algoritmo de enrutamiento*. Estos últimos varían desde los muy simples hasta los extremadamente complejos. Los primeros permiten toma de decisión local más rápida y pueden ofrecer rendimiento total alto a nivel de nodo; sin embargo, con frecuencia no proporcionan adaptabilidad a condiciones cambiantes (como los niveles de tráfico en los *links*) o habilidad para tolerar fallas de recurso. De este modo, la elección de un algoritmo de enrutamiento adecuado conlleva la negociación de ingeniería clásica: simplicidad y bajo costo de implementación frente a mayor adaptabilidad y mejor latencia de peor caso y garantías

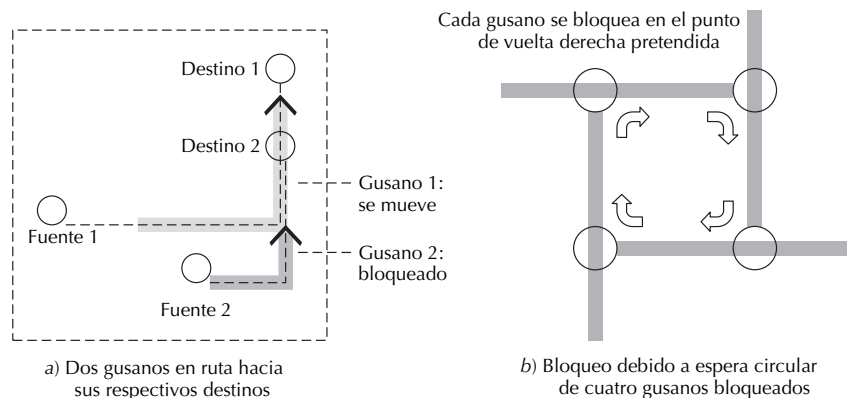


Figura 28.9 Conceptos de conmutación del tipo hoyo de gusano.

de rendimiento total. La conmutación tipo de hoyo de gusano con frecuencia se implementa con enrutamiento simple porque el enrutamiento complejo en parte puede nulificar sus ventajas. Por ejemplo, no es raro tirar cualquier gusano bloqueado, para evitar la cabecera de *buffering* y control. La falta de reconocimiento del nodo de destino la interpreta entonces el emisor como una señal para volver a enviar el mensaje. Si este tipo de tirado y la posterior retransmisión es rara, la rapidez ganada debido a la simplificación puede valer su costo en ancho de banda desperdiciado.

El mejor algoritmo de enrutamiento es dependiente tanto de la topología como de la aplicación. Cuando la ruta tomada por un mensaje depende sólo de los nodos destino y fuente, el algoritmo de enrutamiento es *determinista* u *olvidadizo*. Los algoritmos de enrutamiento *no deterministas*, en los que se elige una de muchas opciones de enrutamiento disponibles, vienen en dos “sabores”. En los algoritmos de enrutamiento *adaptativos*, la elección se puede basar en criterios como disponibilidad de *link* o carga de tráfico (para evitar falla o rutas congestionadas y, por lo mismo, lograr latencia más baja). Los algoritmos *probabilistas* o *aleatorios* se usan para engendrar distribución de tráfico más justo o más uniforme a través de los canales de comunicación disponibles. Con el enrutamiento olvidadizo, la única ruta a seguir por el mensaje se puede determinar en el nodo fuente y la información acerca de él incluida en el mensaje. Tal esquema de *enrutamiento basado en fuente* obvia la necesidad de lógica de decisión en los nodos intermedios, ello conduce a simplificación de hardware y aceleración de enrutamiento. En el otro extremo, a cada nodo o conmutador intermedio sólo se le permite decidir el siguiente nodo o conmutador (conectado a uno de sus canales salientes) en ruta al destino. Un ejemplo de tal esquema es el *enrutamiento de primera hilera* o las redes malla 2D o toro (figuras 28.5a y 28.6a), en el que un mensaje se envía primero a lo largo de una hilera hacia la columna apropiada y luego se enruta arriba o abajo por la columna hasta que alcanza el nodo destino. La decisión se puede basar en un algoritmo simple que se ejecute de manera aleatoria. En forma alterna, las opciones de *link* salientes para alcanzar cada nodo destino pueden precalcularse y almacenarse en *tablas de enrutamiento* para acceso rápido.

Luego que se elige el tipo de mecanismo que pasa mensajes y un algoritmo de enrutamiento asociado, se debe tratar con la emisión de manipulación de mensaje al nivel software (sistema operativo). Un proceso de usuario inicia mensajes al pedir rutinas de sistema operativo relevantes, mucho como en la realización entrada/salida. Por cada mensaje enviado, el sistema operativo asigna espacio de *buffer*, realiza las verificaciones requeridas y establece los parámetros de mensaje relevantes antes de inyectarlos en la red. En el lado receptor, similar proceso ocurre a la inversa. Las partes de mensaje se reciben, ensamblan en un *buffer* y verifican. En este punto el mensaje ha llegado pero el proceso usuario todavía no lo envía al destino. La llegada de un mensaje puede generar entonces una interrupción que conduzca a que el proceso usuario adecuado sea notificado o alertado. Este involucramiento por el software de sistema operativo causa que la iniciación y recepción de mensajes sean procesos relativamente lentos. No es raro que la cabecera se acerque, o incluso supere, 1 ms. Por esta razón, la latencia de mensaje real en el nivel de hardware es mucho menos crucial de lo que fue en el caso para multiprocesadores de memoria distribuida. Sin embargo, el rendimiento total de red sigue siendo importante porque refleja la capacidad de comunicación global.

■ 28.4 Construcción y uso de multicomputadoras

Como es el caso con los multiprocesadores asimétricos (memoria distribuida), es posible implementar una multicomputadora con el uso de un bus compartido como red de interconexión. Se pueden usar múltiples buses paralelos para resolver el problema de ancho de banda y para asegurar operación continua ante malos funcionamientos de bus. Por ejemplo, se puede vislumbrar 16 procesadores conectados a cuatro buses diferentes, de modo que hay cuatro rutas de comunicación alternas desde un procesador hasta otro. Conforme crece el número de éstos, se pueden usar buses jerárquicos, como los de la figura

28.7a (de nuevo con más de un bus en cada nivel, por razones de rendimiento y confiabilidad). Observe que aquí no hay espacio de dirección compartido, así como ninguna comunicación para forzar coherencia de datos. De este modo, el tráfico sobre los buses consiste por completo de mensajes explícitos enviados entre procesos. El volumen de tales mensajes es dependiente de aplicación pero, en general, tiende a ser menor que la comunicación de datos requerida en un multiprocesador de memoria compartida.

Las multicomputadoras se usan en dos modos distintos. Primero las computadoras múltiples pueden ejecutar tareas independientes. En este modo independiente, el multicomputador actúa como un conjunto de máquinas simples que comparten I/O, almacenamiento y otros recursos. Por ejemplo, en un sitio de motor de búsqueda de Internet se pueden emplear miles de computadoras de tipo PC, y cada consulta se asigna para procesamiento a una de las computadoras disponibles. Tales consultas son completamente independientes y no interactúan unas con otras excepto cuando acceden a archivos e índices de datos compartidos; los últimos usualmente se replican para facilitar conflictos. Segundo, se pueden requerir los nodos de una multicomputadora para resolver un problema que tiene importantes requerimientos computacionales. En este modo cooperativo, el problema de interés se debe dividir en un conjunto de *procesos de comunicación*. Los mecanismos para comunicación se discutieron antes en este capítulo. Lo que resta para la implementación adecuada de una multicomputadora es una instalación, en el nivel del sistema operativo, para permitir la asignación de tareas a los nodos, vigilancia del progreso de cada tarea, prevención de bloqueos (espera circular) y generalmente asegurar que el cálculo global se complete rápida y eficientemente en términos o utilización de recursos.

La calendarización (*scheduling*) de tareas es difícil. Un ejemplo de un conjunto de tareas de comunicación, o un *sistema de tarea*, se muestra en la figura 28.10. Cada bloque en el diagrama representa una tarea indivisible, con su tiempo de ejecución t . Las flechas indican comunicación, con la suposición de que la comunicación ocurre después de que la tarea en la cola de la flecha se completó y antes de que la tarea en la punta de la flecha pueda comenzar ejecución (por simplicidad se ignora la latencia de comunicación). Este tipo de dependencia es similar a las relaciones de cursos en una universidad en términos de prerrequisitos. La figura 28.10b muestra cómo se puede calendarizar el sistema de tareas de la figura 28.10 en uno, dos o tres nodos de computación para respetar las dependencias de prerrequisitos (por ejemplo, tanto B como C se completaron antes que F iniciara) y para completar el sistema de tareas en el tiempo mínimo posible. Observe que es imposible ejecutar el sistema de tareas de la figura 28.10 en menos de siete unidades de tiempo, de modo que usar más de tres nodos no ayudaría a reducir el tiempo de ejecución total. Esta es una manifestación de la ley de Amdahl, como se discute en el ejemplo siguiente.

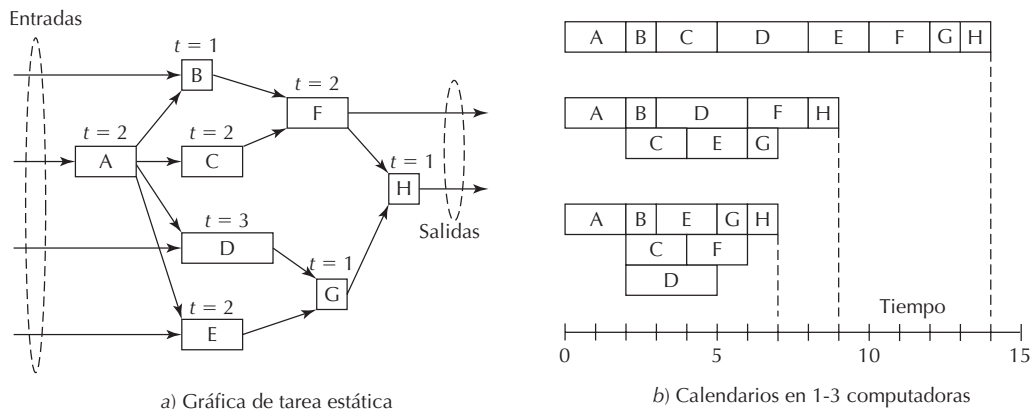


Figura 28.10 Un sistema de tareas y calendarios en una, dos y tres computadoras.

Ejemplo 28.1: Calendarización de tarea y ley de Amdahl Considere el sistema de tareas de la figura 28.10a. Las tareas A y H no se pueden ejecutar en paralelo con cualquier otra tarea, ello implica que el sistema de tareas tiene una fracción inherentemente secuencial de $f = 3/14$ (tres unidades de tiempo de 14 en total). La ley de Amdahl sugiere que la aceleración alcanzable está acotada de manera superior por $14/3 = 4.67$. Demuestre que, para este caso, debido a la suposición de tareas indivisibles, la aceleración está acotada superiormente por 2. Entonces, formule una alternativa a la ley de Amdahl que produzca esta cota más apretada.

Solución: Al examinar las gráficas de calendarización de la figura 28.10b, se observa que las tareas A y H “se pegan” a los dos extremos, y el paralelismo está limitado a las tareas intermedias. La ley de Amdahl sugiere que, con aceleración infinita en esta parte paralela, al llevar su tiempo de ejecución a 0, el tiempo de ejecución total se reducirá de 14 a 3, para una aceleración de $14/3$. Esto último requeriría que las tareas se trocen en piezas pequeñas que se puedan ejecutar en paralelo. Sin embargo, en virtud de que la tarea D es indivisible, su ejecución requiere tres unidades de tiempo, sin importar cuántas computadoras se usen. De hecho, puesto que D y G se deben ejecutar en secuencia (lo cual requiere cuatro unidades de tiempo), ninguna cantidad de paralelismo puede reducir el tiempo de ejecución de las tareas entre A y H a menos de cuatro unidades de tiempo. Por tanto, el mejor tiempo de ejecución posible es $3 + 4 = 7$, para una aceleración de $14/7 = 2$, que ya se logró con tres computadoras. De manera general, cada gráfica de tarea tiene una o más *rutas cruciales*, definidas como la ruta de ejecución más larga para una cadena de nodos entre las entradas y las salidas. En la gráfica de tareas de la figura 28.10a, la ruta crítica consiste de A, D, G y H, con un tiempo de ejecución total de siete unidades. Por tanto, una cota superior sobre la aceleración representa la suma de tiempos de corrido para todas las tareas, dividida entre la longitud de la ruta crítica ($14/7 = 2$ en el ejemplo).

La discusión de los sistemas de tareas y la calendarización fue enormemente simplificada. La siguiente constituye una lista parcial de factores que hacen más difícil encontrar un calendario óptimo para un sistema de tareas en una multicomputadora.

Los tiempos de ejecución de tareas no son constantes (son dependientes de datos).

Todas las tareas se desconocen *a priori* (las tareas se pueden generar o eliminar dinámicamente).

Se debe factorizar la cabecera de tiempo de comunicación.

El paso de mensaje no ocurre en límites de tarea.

Es posible que no todo nodo en la multicomputadora sea capaz de ejecutar todas las tareas.

Con frecuencia se toma un enfoque de dos gradas a este problema dinámico. Una asignación de tareas a los nodos de computación se realiza al arranque, y el progreso de cada tarea vigila su comportamiento de comunicación. Si algunos nodos están mucho más cargados que otros, o si la distribución de tareas es tal que la comunicación excesiva frena el sistema, las tareas se pueden reasignar como parte de un proceso de *equilibrio de carga*.

Una estrategia común para construir multicomputadoras es a través de aplicar computadoras modulares autocontenidas en anaqueles e interconectarlas mediante tecnología de red (figura 28.11a). A los sistemas resultantes con frecuencia se les refiere como *clusters*. Con el adecuado hardware y software de avanzada para distribución de trabajo y equilibrio de carga, tal organización se puede escalar hasta tamaños muy grandes para ciertos tipos de carga de trabajo comúnmente encontrada en comercio electrónico y aplicaciones de bases de datos. Hardware compacto, junto con avances en conectividad inalámbrica, permitirán un enfoque de composición de bloques de juguete para construir grandes multicomputadoras en lo futuro (figura 28.11b).

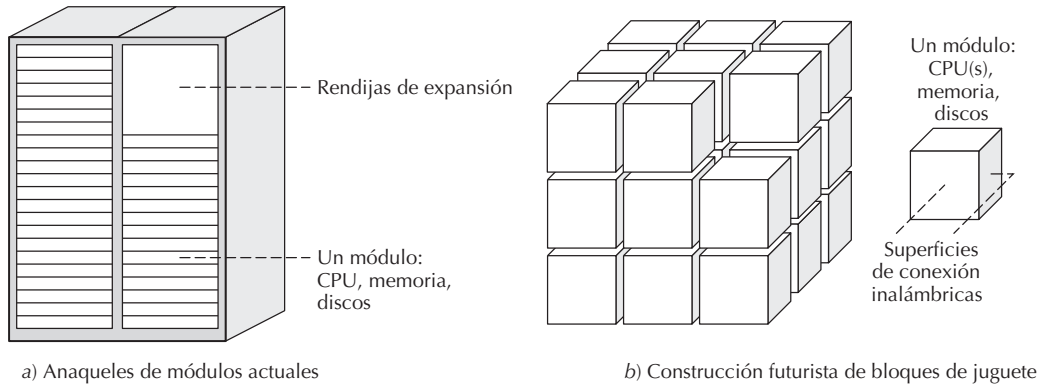


Figura 28.11 Clusters crecientes que usan nodos modulares.

■ 28.5 Computación distribuida basada en red

Las multicomputadoras consideradas hasta el momento en este capítulo tienden a tener nodos de computación idénticos o similares y a conectarse mediante interconexiones bastante regulares de los tipos que se muestran en las figuras 28.6 y 28.7. Tal uniformidad y regularidad simplifican la sustitución de un nodo fallido (se necesitan pocos componentes de repuesto), calendarización de tareas en nodos de computación (pues cada nodo es capaz de ejecutar cada tarea) y otras numerosas operaciones. Sin embargo, tal estructura impide tanto la escalabilidad simple como el uso de una mezcla de recursos de propósitos general y especial para resolver cooperativamente grandes problemas. En consecuencia, el área de computación distribuida basada en red ha recibido considerable atención en las pasadas dos décadas.

Además de la expansión simple a través del agregado de más nodos de computación y ancho de banda de red, un importante beneficio colateral de este enfoque es la capacidad de juntar inactividad computacional, almacenamiento y recursos de comunicación para usar como supercomputadora virtual. Para ver esto, recuerde que virtualmente toda computadora personal se pone en modo inactivo la mayor parte del tiempo, las unidades de disco en promedio están menos que medio llenas y el tráfico de datos sobre la mayoría de los *links* de red consumen una pequeña fracción de sus anchos de banda pico. De hecho, para algunos dominios de aplicación limitados, estos recursos inactivos ya se utilizan para formar supercomputadoras virtuales multiteraflops. SETI@home, una organización dedicada a investigar inteligencia extraterrestre, explotó exitosamente este enfoque, proporcionando los programas y datos requeridos a participantes voluntarios y utilizando sus capacidades computacionales inactivas para analizar datos tomados de radiotelescopios para signos de vida inteligente en el espacio exterior.

La computación distribuida con base en la red se usa en muchas formas. Cuando la red es una red de área local (LAN) con extensión limitada y los nodos de computación representan estaciones de trabajo o computadoras personales, a los sistemas distribuidos resultantes a veces se les denomina *redes* o *clusters de estaciones de trabajo* (NOW, COW). Usualmente, un *cluster* se usa para un conjunto de nodos que se pretende trabajen juntos y no tengan identidades externamente distintas, mientras que los nodos en una red de estaciones de trabajo se puede usar y direccionar externamente por separado. Cuando se usa una red de área ancha (en el extremo, Internet) para comunicación entre nodos de computación, el sistema resultante es una *metacomputadora*. Los nodos computacionales en la computación distribuida son computadoras completas, cada uno con su propia capacidad entrada/salida y sistema operativo. Con frecuencia a éstos se les conoce como *sistemas* o *plataformas finales*. Es común caracterizar una

plataforma (aunque incompletamente) al nombrar su arquitectura de procesador y sistema operativo. La plataforma más comúnmente usada hoy día es Intel-x86/Windows. Sun-SPARC/SunOS, Intel-x86/Linux y PowerPC/MacOS son otros ejemplos comunes de plataformas o sistemas finales.

Para juntar recursos computacionales conectados a una red de área local o ancha para resolver grandes problemas o muchas instancias de problemas más pequeños, se requiere una estrategia de coordinación. La estrategia difiere de acuerdo con el modelo computacional a usar. En una arquitectura *cliente-servidor*, las solicitudes de servicio de los procesos cliente se envían a servidores especializados que realizan el servicio requerido o se arreglan para que se realice en otra parte. En esta arquitectura, puede haber múltiples servidores para un tipo particular de servicio y algunos servidores pueden tener *proxies* que reduzcan la carga sobre los servidores originales y sobre la red al guardar en caché objetos de datos de reciente acceso (como es común para los servidores web). Las variaciones de la arquitectura cliente-servidor incluyen *código móvil* y *agentes móviles*, que transfieren la computación a sitios más cercanos a los datos requeridos y otros servicios con la intención de reducir la carga de comunicación y el retardo. En este contexto, en una arquitectura *par a par*, los procesos que corren en diferentes nodos interactúan como pares. Éste es el modelo apropiado para que muchos nodos computacionales resuelvan un gran problema cooperativamente. Tales procesos necesitan rutinas de sistema, a las que comúnmente se les refiere como *middleware*, para ayudar a coordinar sus acciones. Por ejemplo, se puede instrumentar una instalación de *grupo de comunicación* para notificar a todos los procesos participantes los cambios en el entorno computacional.

El uso de instalaciones de nodos de computación y comunicación comerciales con el modelo de interacción par a par es una elección natural desde los puntos de vista costo, facilidad de implementación y mantenimiento. Sterling argumenta [Ster01] que para cada MFLOPS de potencia computacional, una supercomputadora idónea puede costar diez veces más que adquirir una PC estándar. Y esto resulta antes de que se tomen en cuenta costos de uso y mantenimiento. La figura 28.12 muestra la estructura de una red de estaciones de trabajo. La interfaz de red contiene un procesador dedicado y una memoria bastante grande (SRAM, por sus siglas en inglés), que es capaz de alimentar mensajes hacia, o recibirlos desde, la red de alta rapidez. Por ejemplo, se pueden emplear múltiples canales DMA en la unidad de interfaz de red para inyección de mensaje, expulsión de mensaje y transferencia de mensaje hacia/desde la PC anfitriona.

Además del software de manejo de mensaje que corre en la interfaz de red, también se necesita un sistema de coordinación de nivel superior. Este sistema representa un sistema operativo distribuido que con mucha frecuencia no se construye desde cero, sino que más bien se forma al agregar otra capa de capacidades a un sistema operativo estándar. Entre las capacidades que se pueden proporcionar mediante la extensiones se encuentran:

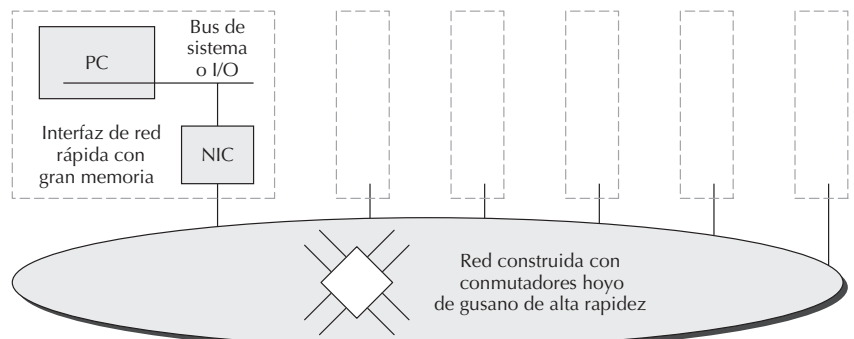


Figura 28.12 Red de estaciones de trabajo.

Mantenimiento de información del estatus de sistema para uso eficiente y recuperación.

Fraccionamiento y calendarización de tareas.

Asignación de tarea inicial a nodos (y, quizá, equilibrio de carga dinámico).

Gestión de sistema de archivo distribuida.

Compilación de datos de uso del sistema con propósitos de planeación y recuento.

Reforzamiento de privacidad intersistema y políticas de seguridad.

Conforme se extienda más el uso de la computación distribuida basada en red, algunas o todas estas funciones serán incorporadas en los sistemas operativos estándar, ello hará más sencillo configurar y usar redes de estaciones de trabajo.

■ 28.6 Computación en retícula y más allá

El modo actual de usar los recursos de cómputo, es decir, tener una computadora en cada lugar donde haya necesidad de cómputo, es similar a instalar un generador eléctrico en cada hogar u oficina. Tal como una retícula de energía eléctrica permite un desacoplamiento entre el sitio donde se genera la energía y donde se usa, una *retícula de computación* permite la distribución de capacidades computacionales a los usuarios sin importar dónde se origina la potencia de procesamiento. La computación en retícula, en su forma ideal, permite a los usuarios conectar dispositivos simples, que consisten de interfases de usuario adecuadas, en contactos de pared que hacen que la potencia de computación esté fácilmente disponible a un costo nominal. Tal como la energía eléctrica se vende a cierto precio por kilowatt-hora, la energía de computación se puede cobrar en, por decir, GFLOPS-hora. Otros recursos relacionados con la computación, como el espacio de almacenamiento de archivo, el software de aplicación y el servicio remoto de atención de problemas, se pueden ofrecer de modo similar a los usuarios, quienes entonces pagarán mensualmente una factura por todos los servicios de computación usados.

El escenario idealizado en el párrafo anterior no se ha materializado, pero los investigadores de la computación en retícula trabajan en varias herramientas necesarias para integrar recursos de computación heterogéneos en un sistema uniforme con muchos usuarios diversos. El conjunto de recursos de hardware y software así proporcionados aparecen a los usuarios como una *metacomputadora* con amplias capacidades computacionales. Las múltiples metacomputadoras lógicas vistas por los diferentes usuarios pueden traslaparse en su uso de recursos.

La computación en retícula es significativamente diferente de la computación de tiempo compartido, que era muy común en las décadas de 1960 y 1970. La compartición de tiempo estuvo motivada principalmente por la escasez y alto costo de las computadoras y permitió a muchos usuarios compartir los costos de adquisición y mantenimiento de los recursos de computación. La computación en retícula, por otra parte, busca proporcionar servicios confiables y ubicuos virtualmente para todos los usuarios. Las ventajas de la computación en retícula son similares, pero no limitadas, a las de las retículas de electricidad:

Disponibilidad casi continua de recursos de computación y relacionados.

Requerimiento de recursos con base en suma de promedios, en lugar de por suma de picos.

Pago por servicios basados en uso real en lugar de por demanda pico.

Almacenamiento de datos distribuidos para mayor confiabilidad, disponibilidad y seguridad.

Acceso universal a recursos de computación especializados y “uno en su tipo”.

Una retícula de computación abarca una jerarquía de sistemas, desde los sistemas finales o plataformas, a través de *clusters* de nodos, hasta los intranet e internet. Cada uno de estos niveles existe actual-

mente en forma bastante evolucionada como resultado de muchos años de investigación y desarrollo y uso extenso. Sin embargo, la integración exitosa de dichos niveles en una retícula de computación requiere mayor énfasis en los requisitos de interacción para aplicaciones de retícula usuales, que son muy diferentes de la forma en que los sistemas interactúan en la actualidad.

Las consideraciones anteriores afectan todos los niveles de la jerarquía de la retícula, desde los sistemas finales hasta las Internet.

1. Al nivel de sistema final o plataforma, se debe poner más énfasis en las capacidades de comunicación e interfases, mejorar el rendimiento de comunicación y permitir integración más sencilla de los nodos en los *clusters*.
2. Al nivel de *cluster*, se necesitan mejoras tanto en capacidades hardware como en el software de sistema operativo para soportar coordinación de *cluster* y comunicación con el mundo exterior. En particular, la convergencia entre el software de sistema final y software de nivel *cluster* es una tendencia bienvenida.
3. Las intranets abarcan recursos de computación independientes con control centralizado (organizacional) limitado, aunque prácticamente insignificante. El enfoque actual en las intranets es la flexibilidad y la facilidad de expansión. Se necesita mejorar el rendimiento de comunicación mediante el uso de interacción más ligera, en gran parte como en los *clusters*.
4. Usualmente las internet no tienen control centralizado. Otros factores que complican incluyen amplia distribución geográfica que conducen a tiempo de propagación de señal no despreciable, cumplimiento de seguridad, y conflictos en intercambio de datos tanto entre organizaciones como a nivel internacional.

Actualmente se persiguen tres enfoques complementarios [Fost99]. El primero se apoya en los componentes mercantiles y los usos de la llamada arquitectura de tres gradas, en la que los servidores de aplicación de media grada median entre servicios *back-end* sofisticados (como grandes bases de datos) y *front-end* relativamente simples. El segundo enfoque se basa en un diseño orientado a objeto donde los recursos de computación son objetos anfitriones, los recursos de almacenamiento son objetos de bóveda de datos, etcétera, cada uno con sus propiedades y métodos de acceso específicos. Esto último se ejemplifica mediante el sistema Legion. El tercer enfoque, seguido por el sistema Globus, toma la visión de que la tecnología mercantil actual es inadecuada y que la arquitectura en retícula debe proporcionar servicios básicos sin restringir el modelo de programación. En concordancia, este enfoque favorece el desarrollo de un conjunto de herramientas de servicios de bajo nivel para seguridad, comunicación, ubicación de recursos, asignación de recursos, gestión de procesos, acceso a datos, y cosas parecidas.

Si se despliega exitosamente, una retícula de computación permitirá supercomputación distribuida (reunión de las potencias computacionales de muchos nodos para atender una aplicación), computación de alto rendimiento total, computación bajo pedido, computación de datos intensos (análisis y asimilación de grandes cantidades de datos que están geográficamente distribuidos) y computación colaboradora.

PROBLEMAS

28.1 Paso de mensaje

En una multicomputadora de 64 procesadores, interconectada como una malla 8×8 , se realizará el siguiente cálculo de diferencia finita 2D. el cálculo progresa en fases, donde en cada fase, todo elemento $A_{i,j}$ de una ma-

triz 1024×1024 se debe actualizar de acuerdo con los valores de sus cuatro vecinos $A_{i,j-1}$, $A_{i,j+1}$, $A_{i-1,j}$, y $A_{i+1,j}$ al final de la fase precedente, donde existen estos elementos. Por ende, actualizar cada elemento requiere acceso a cuatro valores de ocho bytes que pueden estar

disponibles localmente o desde un nodo vecino. Considere dividir la matriz en submatrices 32×32 que se almacenan en los nodos de la malla 8×8 en el orden natural.

- Calcule la cantidad total de intercambio de datos entre nodos si el cálculo continúa durante mil fases.
- ¿Cuál es el número total de pasos de comunicación si un procesador puede enviar un mensaje a sus cuatro vecinos en un paso?
- Repita la parte b), pero suponga que los mensajes a los cuatro vecinos se deben enviar uno a la vez.
- Demuestre que el fraccionamiento sugerido de la matriz es el mejor posible, en lo concerniente a la cantidad de intercambio de datos.

28.2 Memoria buzón

Una *memoria buzón* (*mailbox*) constituye una RAM que tiene un bit *flag* lleno/vacío asociado con cada una de sus palabras. Considere una instrucción “put” que especifica un registro, una localidad de memoria y una dirección de destino de bifurcación. La instrucción se ejecuta atómicamente y provoca una de dos acciones: si la localidad de memoria direccionada está vacía, el contenido del registro se almacena en ella y sus *flags* se cambian a “lleno”; de otro modo, no tiene lugar escritura de memoria y el control se transfiere a la dirección de bifurcación especificada.

- Proporcione una descripción adecuada para la instrucción “get” complementaria.
- Discuta cómo se pueden comunicar dos procesos a través de la memoria buzón usando mensajes de una palabra o multipalabra.
- Demuestre cómo se puede lograr el efecto de memoria buzón con una memoria compartida ordinaria con el uso de la instrucción probar y establecer discutida en la sección 27.3.

28.3 Redes de interconexión directa

- Demuestre que el toro 4×4 y el hipercubo de 16 nodos de la figura 28.6 son realmente la misma red.
- Muestre que el anillo cordado de 16 nodos de la figura 28.6 es similar al toro 4×4 , excepto por una conectividad diferente para los *links* de giro final.

28.4 Redes de interconexión directa

- Derive el diámetro de cada una de las redes que se muestran en la figura 28.6.
- Derive el ancho de bisección de cada una de las redes que se muestran en la figura 28.6.
- Generalice los resultados de la parte a) con redes similares de 2^{2q} nodos, donde q es entero. Suponga que la distancia de salto (*skip*) del anillo cordado es 2^q .
- Repita la parte c) en conexión con los resultados de la parte b).
- ¿En qué difieren los anchos de bisección y diámetros de una malla 2D de 2^{2q} nodos y un toro?

28.5 Redes de interconexión directa

Una red de interconexión es fuerte si sus nodos siguen conectados (se pueden comunicar mutuamente) a pesar de fallos de nodo o *link*. Una medida de fuerza para una red es su *conectividad de nodo*, que se define como el número mínimo de nodos cuya remoción de la representación gráfica de la red desconecta algunos nodos de los otros. Una segunda medida de fuerza es la *conectividad de borde*, que se define como el número mínimo de bordes cuya remoción conduce a la desconexión.

- ¿Cuáles son las conectividades de nodo de las redes que se muestran en la figura 28.6?
- ¿Cuáles son las conectividades de borde de las redes que se muestran en la figura 28.6?
- Compare las conectividades de nodo y de borde de una malla cuadrada 2D con las correspondientes al toro.
- Generalice la comparación de la parte c) con las redes malla q D y toro.
- ¿Las conectividades de nodo y borde de una red se relacionan de alguna forma?

28.6 Redes de interconexión indirecta

- Muestre que, en la red omega de la figura 28.6b, cada nodo está conectado a los demás nodos a través de una ruta única mediante cuatro *switches*.
- Muestre que la red mariposa de la figura 27.2a se puede volver a dibujar como la red omega de la figura 28.7b al reenumerar los nodos en sus lados izquierdo y derecho.

28.7 Red de intercambio Waksman

Una red de intercambio Waksman $2^q \times 2^q$ con 2^q entradas y 2^q salidas se construye recursivamente, a partir de *switches* 2×2 , del modo siguiente. Primero, se usan 2^{q-1} *switches* en la columna cero para conmutar las entradas 0 y 1, 2 y 3, etcétera. Las salidas superior e inferior de los *switches* de la columna 0 se permutan por separado al usar dos redes $2^{q-1} \times 2^{q-1}$ construidas en la misma forma. Los *switches* de estas redes formarán las columnas 1 a $2q - 3$ en la eventual red de $(2q - 1)$ etapa. Finalmente, las salidas correspondientes de las dos redes de intercambio de medio tamaño, excepto para sus más bajas, se permutan con el uso de $2^{q-1} - 1$ *switches* en la columna $2q - 2$.

- Muestre que, en total, se usan $2^q q - 2^q + 1$ *switches* en la construcción de la red.
- Dibuje una red de intercambio Waksman 4×4 completa y demuestre que en realidad puede enrutar todas los intercambios.
- Muestre que una red de intercambio $n \times n$ arbitraria, donde n no es una potencia de 2, se puede obtener al podar una red de intercambio Waksman con la siguiente potencia de 2 más larga.
- Aplice el resultado de la parte c) a la construcción de una red de intercambio 6×6 .

28.8 Enrutamiento en multicomputadoras

Si ignora la contención por mensajes múltiples para usar el mismo *link* (es decir, si supone una red muy ligeramente cargada), toma $hk/b + (h - 1)d$ unidades de tiempo enviar un mensaje de tamaño k a lo largo de una ruta de h *links* (h saltos) usando enrutamiento almacenar y adelantar, donde b representa el ancho de banda del *link* y d el retardo de enrutamiento por brinco. La fórmula correspondiente para conmutación del tipo hoyo de gusano es $k/b + (h - 1)d$. Ignore el hecho de que el retardo de enrutamiento d por brinco puede ser diferente dependiendo de la red de interconexión y el algoritmo de enrutamiento empleado. En sus cálculos, suponga $b = 100$ MB/s y $d = 200$ ns.

- Justifique las ecuaciones de retardo de enrutamiento dadas.
- Para cada red en la figura 28.6, calcule el retardo de enrutamiento máximo, por separado, para mensajes cortos y largos de tamaño 256 y 1024 B, respectivamente. Suponga enrutamiento almacenar y adelantar.

- Repita la parte b) para conmutación del tipo hoyo de gusano.
- Calcule las latencias por byte de mensajes cortos y largos en las partes b) y c) y comente los resultados.

28.9 Conmutación del tipo hoyo de gusano

Considere conmutación del tipo hoyo de gusano con mensajes muy cortos que consisten de un *flit* de *header* (que contiene la dirección de destino) y un *flit* de información sencillo.

- En este caso, ¿la conmutación del tipo hoyo de gusano todavía tiene ventajas sobre la conmutación en paquete?
- ¿Es posible bloqueo con tales mensajes cortos?
- Suponga que el *flit* de información sencillo contiene la identidad del nodo emisor. ¿Tiene algún propósito útil un mensaje que sólo contiene las identidades de los nodos fuente y de destino?
- Si va un paso más adelante, ¿tiene sentido que un mensaje contenga sólo la dirección de destino y ningún *flit* de información?

28.10 Transmisión en multicomputadoras

Considere una multicomputadora conectada en malla de $2^q \times 2^q$ y el siguiente algoritmo de transmisión recursiva basado en envío de mensajes punto a punto. Al arranque, el nodo fuente envía cuatro mensajes a los cuatro nodos ubicados en la esquina inferior izquierda de cada uno de los cuatro cuadrantes de la malla. Cada uno de estos cuatro nodos usa entonces el mismo método para transmitir el mensaje dentro de su propio cuadrante.

- Si la transmisión de mensaje a un vecino tarda tiempo unitario y un nodo puede enviar un mensaje sólo a un vecino en cada paso, ¿cuál es el tiempo de transmisión total?
- ¿Cuál es el número total de mensajes enviados entre nodos durante la transmisión?
- Discuta el comportamiento del algoritmo, incluidos los conflictos en enrutamiento, cuando se usa conmutación del tipo hoyo de gusano en lugar de enrutamiento almacenar y adelantar.

28.11 Calendarización de tareas

Considere la gráfica de tareas de la figura 28.10a y sus calendarios asociados en la figura 28.10b. Sin cambiar

los elementos existentes de la gráfica de tareas, agregue una nueva tarea sencilla I a la gráfica, de modo tal que se satisfagan las condiciones dadas en las partes a)-e), o demuestre que satisfacer las condiciones es imposible. Calcule las nuevas cifras de aceleración para cada caso que sea factible.

- No cambian las longitudes de los calendarios de 2 y 3 procesadores (9 y 7).
- El uso de cuatro procesadores produce mayor aceleración que tres de éstos.
- El uso de tres procesadores no produce mayor aceleración que dos de éstos.
- Con tres procesadores se logra una aceleración de al menos 2.5.
- La ejecución de una serie infinita de instancias de la gráfica de tareas en los tres procesadores mantiene a todos los procesadores completa y continuamente ocupados.

28.12 Calendarización de tareas

Considere una gráfica de tareas con una sola tarea de entrada y una tarea de salida (como la figura 28.10a, pero la tarea F no proporciona salida), y que ambas son de tiempo unitario. Entre estas tareas de entrada y salida hay k tareas paralelas con tiempo de ejecución idéntico t , donde t no necesariamente es entero.

- Muestre cómo se calendariza esta gráfica de tareas para correr en p procesadores y encuentre su tiempo de ejecución total. Suponga que k es divisible por p .
- Determine la aceleración resultante a partir del calendario de la parte a). ¿Cuál es el valor mínimo de k para el que se puede lograr una aceleración de s o más?
- Relacione la fórmula de aceleración de la parte b) con la fórmula de aceleración de Amdahl. En particular, encuentre la fracción secuencial f de la fórmula de Amdahl en términos de los parámetros k y t .

28.13 Teorema de calendarización de Brent

Sea T_p el tiempo mínimo necesario para ejecutar una gráfica de tarea particular con p procesadores. El teorema de calendarización de Brent postula que $T_p < T_\infty + T_1/p$, donde T_∞ , el tiempo de ejecución con un número ilimitado de procesadores, es la profundidad (o longitud de ruta crítica) de la gráfica.

- Pruebe el teorema de calendarización de Brent para el caso especial de tareas de tiempo unitario. *Sugerencia:* Considere la rendija de tiempo más temprano en el que cada tarea se puede iniciar y que hay n_i tareas cuyo posible tiempo de inicio más temprano es i ; estas tareas se pueden ejecutar en $\lceil n_i/p \rceil$ unidades de tiempo con el uso de p procesadores.
- Demuestre que, para $p \geq T_1/T_\infty$, se tiene $1 \leq T_p/T_\infty < 2$.
- Demuestre que, para $p \leq T_1/T_\infty$, se tiene $p/2 < T_1/T_p \leq p$.
- Con base en los resultados de las partes b) y c), argumente que $p = T_1/T_\infty$ es, en un sentido, un número adecuado de procesadores a usar para ejecutar una gráfica de tareas.

28.14 Clusters y redes de estaciones de trabajo

Una red de estaciones de trabajo consta de computadoras personales en un número de oficinas adyacentes vinculadas por cables ópticos de 10 m de largo, en los que la luz viaja a dos tercios de la rapidez de la luz en espacio libre.

- si cada estación de trabajo ejecuta instrucciones a la tasa de 2 GIPS, ¿cuántas instrucciones se ejecutarán durante el tiempo de viaje de la señal entre oficinas adyacentes?
- ¿Qué tipo de límite impone la observación en la parte a) sobre la granularidad de la computación paralela en tal sistema de computación distribuida?

28.15 Variables compartidas frente a paso de mensaje

Considere el problema de convertir un programa paralelo escrito por un multiprocesador escrito para un multiprocesador de memoria compartida que se puede ejecutar sobre una multicomputadora de paso de mensaje y viceversa. ¿Cuál de las dos conversiones diría es más simple y por qué?

28.16 Proyecto Deep Gene de IBM

A finales de 1999, IBM anunció un proyecto de investigación y desarrollo denominado código “Deep Gene”, designado para conducir a la primera supercomputadora de clase PFLOPS para acometer el enormemente desafiante problema computacional de desdoblamiento

de proteínas, que es de interés para las comunidades de investigación médica y farmacéutica. De acuerdo con informes preliminares, el diseño es usar un millón de “simples” procesadores GFLOPS sin caché, integrados con memoria, 32 por chip, ocho hilos por procesador, 64 chips en cada tarjeta de $60 \times 60 \text{ cm}^2$ (2 TFLOPS), ocho tarjetas en un anaquel de 1.8 m de alto (16 TFLOPS), 64 anaqueles, que ocupan una área aproximada de 200 m^2 . Estudie este proyecto y escriba un informe de cinco páginas que se enfoque en sus características arquitectónicas.

28.17 Computación en retícula

Un conflicto que se debe resolver antes de que se puedan vender los servicios de computación (en la misma forma que la electricidad, el gas natural y el servicio telefónico) consiste en investigar el consenso acerca de una unidad adecuada para cobrar el servicio (similar a kilowatt-hora, metros cúbicos y minutos de conectividad para instalaciones comunes). Estudie este problema y escriba un informe de cinco páginas que destaque y compare las diversas propuestas a este respecto.

REFERENCIAS Y LECTURAS SUGERIDAS

- [Coul01] Coulouris, G., J. Dollimore y T. Kindberg, *Distributed Systems: Concepts and Design*, Addison-Wesley, 3a. ed., 2001.
- [Cull99] Culler, D. E. y J. P. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1999.
- [Duat97] Duato, J., S. Yalmanchili y L. Ni, *Interconnection Networks: An Engineering Approach*, IEEE Computer Society, 1997.
- [Fost99] Foster, I. y C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1999.
- [Harg01] Hargrove, W. W., F. M. Hoffman y T. Sterling, “The Do-It-Yourself Supercomputer”, *Scientific American*, vol. 285, núm. 2, pp. 72-79, agosto de 2001.
- [Hord93] Hord, R. M., *Parallel Supercomputing in MIMD Architectures*, CRC Press, 1993.
- [Mile03] Milenkovic, M., *et al.*, “Toward Internet Distributed Computing”, *IEEE Computer*, vol. 36, núm. 5, pp. 38-46, mayo de 2003.
- [Parh99] Parhami, B., *Introduction to Parallel Processing: Algorithms and Architectures*, Plenum Press, 1999.
- [Snir96] Snir, M., *et al.*, *MPI: The Complete Reference*, MIT Press, 1996.
- [Ster01] Sterling, T., “How to Build a Hypercomputer”, *Scientific American*, vol. 285, núm. 1, pp. 38-45, julio de 2001.
- [WWW] Nombres Web de algunos de los fabricantes de multicomputadoras: fujitsu.com, hp.com, ibm.com, sgi.com, sun.com.

ÍNDICE

“Si no lo encuentra en el índice, búsquelo cuidadosamente en todo el catálogo.”
Guía del consumidor, Sears, Roebuck, and Co., 1897.

Los conceptos de los problemas que se encuentran al final del capítulo no fueron incluidos en el índice, salvo que introdujeran referencias elementales. Los apellidos y nombres propios de personas se incluyen sólo cuando forman parte de algún término.

- ±0, 172-73, 224-226
- \$0-\$31 (registros MiniMIPS), 85
- ±¥, 172-173, 224-226
 - aritmética, 219-239
 - bifurcación condicional, 232
 - comparación, 232-233
 - conversión, 231
 - distribución de números, 220
 - división, 229
 - error, 233-236
 - estándar ANSI/IEEE, 172-173
 - excepción, 224-226
 - exponente, 172-173
 - formato largo, 172-173
 - instrucción, 230-233
 - multiplicación, 2
 - número desnormalizado, 173
 - pérdida de significancia, 227, 235
 - sesgo de exponente, 173
 - suma, 226-228
 - sumador, 227-228
- ±0, 172-173, 224-226
- 2 por 2, 18, 35, 531
- 80x86, 49, 146-148
- 8086, 49, 148
- Abstracciones en programación, 55
- Abstracciones para I/O, 450-451
- Acarreo
 - aniquilamiento (matar), 181
 - detección de consumación, 195
 - generación, 180-183, 186
 - operador, 185-186
 - propagación, 180-183, 186
 - recurrencia, 181
 - red Brent-Kung, 186-187
 - red de propagación, 180-183
 - salto, 183-182
 - transferencia, 182
- Acceso directo a memoria. *Vea* DMA
- Acceso uniforme a memoria, 509
- Accesos de disco fuera de orden, 360
- Aceleración de hardware, 479-482
 - impacto de rendimiento, 468
- Aceleración encauzada, 284
- Aceleración lineal, 66
- Aceleración, fórmula de Amdahl, 65
- Acelerador de hardware configurable, 480
- Actuador, 355, 405-407
- Actuador de máquina expendedora, 10
- Acumulación de errores, 234
- Acumulador (registro), 142
- Add(ition) (suma)
 - almacenamiento de acarreo, 164, 202-203
 - punto flotante, 226-228
- Agente móvil, 541
- AGP, 444
- Agrupamientos crecientes, 540
- Álgebra booleana, 7
- Algoritmo
 - coherencia de caché, 512-523
 - difusión, 511-512, 532
 - división, 206-215
 - enrutamiento, 530, 535-537
 - multiplicación, 197-206
 - sustitución, 33, 347, 379-385
- Algoritmo de coherencia, 512-523
- Algoritmo de reducción de suma, 512
- Algoritmo de ruteo, 530, 535
 - adaptativo, 537
 - aleatorio, 537
 - almacenar y adelantar, 536
 - basado en fuente, 537
 - bloqueo, 536
 - determinista, 537
 - olvidado, 537
 - paquete, 530-535
 - primero renglón, 537
 - probabilístico, 537
- Almacenamiento desfasado, 496
- ALU (unidad aritmética/lógica), 157-239
- ALU multifunción, 191-193
- ALU, 247
 - archivo de registro, 247
 - bifurcación, 249-250
 - ciclo sencillo, 247-255
 - conjunto de instrucciones, 244
 - decodificador de instrucción, 252
 - ejecución de instrucción, 246-247, 260-261
 - excepciones, 271-273
 - formato de instrucción, 245
 - jumping*, 249-250
 - lectura (*fetch*) de instrucción, 260
 - lógico de dirección siguiente, 249-250
 - máquina de estado de control, 264-266, 272
 - multiciclo, 258-273
 - reloj, 253, 259, 261-262
 - rendimiento, 253-255, 266-267
 - señales de control, 251-253, 262-264
- AMD (American Micro Devices), 148
- American Standard Code for Information Interchange (Código Estándar Americano para Intercambio de Información). *Vea* ASCII
- Analogía con rendimiento de aeronave, 60-61
- Analogía de cajón de escritorio para caché, 339
- Ancho de banda
 - bus, 434-435, 438, 442, 515
 - disco, 362, 423
 - enlace de comunicación, 53-54
- Ancho de bisección, 534-535
- AND
 - compuerta, 4
 - inmediata, 90
 - instrucción, 89-90
- Antifusible, 13
- Aproximación de función, 235-236
- Aproximación polinomial, 235-236
- Apuntador, 113-116
- Apuntador de marcos, 109-110
- Apuntador global, 107
- Actualización, 114-116
 - de marco, 109-110
 - de pila, 107-110
 - global, 107
- Arbitraje, 435, 438-439
- Árbitro de prioridad de rotación, 438
- Árbitro de prioridad fija, 438
- Árbitro, 438. *Vea también* Arbitraje
- Árbol de sumadores con almacenamiento de acarreo, 203
- Árbol multiplicador, 202-203
- Archivo
 - fuerza, 124
 - futuro, 310
 - historia, 310
 - objeto, 124
 - programa de prueba de sistema, 413
 - registro, 28-29
- Archivo de historia, 310
- Archivo estructurado históricamente, 364
- Archivo fuerza, 124
- Archivo futuro, 310
- Archivo objeto, 124
- Aritmética
 - certificable, 235
 - cableada, 472
 - corrimiento. *Vea* Corrimiento
 - desbordamiento, 273
 - exacta, 235
 - instrucción, 89-90
 - intervalo, 235
 - promedio, 73
 - punto flotante, 219-239
 - saturación, 472
- Aritmética certificable, 235
- Aritmética de intervalo, 235
- Aritmética de saturación, 472
- Aritmética envuelta, 472
- Aritmética exacta, 235
- Arquitecto, 40
- Arquitectura
 - avanzada, 465-547
 - cliente-servidor, 541
- Arquitectura avanzada, 465-547
 - encauzamiento avanzada, 306-309
- Arquitectura cargar/almacenar, 84, 98
- Arquitectura cliente-servidor, 541
- Arquitectura de memoria sólo de caché, 521
- Arquitectura Harvard, 345
- Arquitectura procesador en memoria, 332
- Arquitectura Von Neumann, 345

- Arreglo, 113-116
 - disco, 361-365
 - multiplicador, 202-203
 - registro de estado, 501-502
- Arreglo de compuertas programables por campo, 31-32
- Arreglo redundante de discos no costosos, 361-365
- Arreglos de compuertas lógicas, 6
- ASCII (código), 111-112
 - cadena, 127
 - cadena terminada en nulo, 127
 - número, 163
 - tabla, 112
- Asignación de estado, 25
- Asíncrono
 - bus, 435-436
 - diseño, 469
 - modo de transmisión (ATM), 430-432
- Asociatividad en caché, 347-348
- AT adjunto (ATA), 444
- Autoselección, 439, 442
- Barrera de sincronización, 517
- BCD (decimal codificado en binario), 162-163
 - empaquetado, 162
- BCD a siete segmentos, 9
 - instrucción, 252, 265
- Benchmark* (programa de prueba), 68-69
- Benchmarking* (programación de prueba), 68-69
- Beneš, 510
- Bifurcación, 302-303
- Bifurcación condicional. *Vea* Bifurcación
- Bifurcación retardada, 283, 303
- Binario
 - conversión a decimal, 165, 171
 - división, 207-209
 - multiplicación, 198-200
 - punto, 169
 - sumador completo, 164
- BIOS, 450
- Bit
 - cadena, 110
 - manipulación, 90, 191
 - sucio, 343
 - válido, 341, 343
- Bit de uso, 374, 379-380
- Bit sucio, 343, 374, 381
- Bit válido, 341, 343, 374-375, 513
- Bits de permiso, 375-376
- Bloque lógico configurable, 31-32
- Bloqueo en enrutamiento, 536
- Bloques de traslapamiento, 194
- Branch* (bifurcación)
 - hacia adelante, 304
 - hacia atrás, 304
 - igual, 93-94
 - instrucción, 93-96
 - menos que cero, 93-94
 - punto flotante, 232
 - ranura de retardo, 283, 303
 - retardada, 283, 303
 - riesgos, 302-303
- Brecha de rapidez, 330
- Buffer*, 5, 101
- Buffer* auxiliar para traducción de direcciones. *Vea* TLB
- Buffer* de anillo, 101
- Buffer* de inversión, 5
- Buffer* de marcos, 398-399
- Buffer* tres estados, 5
- Buffering* doble, 496-497
- Burbuja (en entrada de compuerta), 4
- Burbuja (en pipeline), 281-283
 - inserción, 281-282
- Bus, 433-440
 - adaptador, 394
 - ancho de banda, 515
 - arbitraje, 438-439
 - asíncrono, 435-436
 - jerárquico, 534
 - protocolo de intercambio, 436
 - trasero, 336
- Bus de interconexión de componente periférico. *Vea* Bus PCI
- Bus de legado, 434
- Bus de sistema, 394
- Bus de transacción dividida, 516
- Bus jerárquico, 534
- Bus multinivel, 534
- Bus PCI, 436-438, 442
- Bus propietario, 434
- Bus serial universal, 442-443
- Bus síncrono, 436
- Bus trasero, 336
- Búsqueda de mayor rendimiento, 74-76
- Búsqueda de membresía, 506
- Búsqueda del valor máximo, 99
- Byte, 85-86, 112
- Byte sin signo, 112
- Cabeza borradora, 365
- Cabeza lectora/grabadora, 354, 365
- Cable coaxial, 432
- Cable módem, 53
- Caché, 330, 335-348
 - ancho de línea, 337, 347
 - asociatividad, 347-348
 - asociativo, 342
 - coherencia, 512-523
 - conexión de memoria principal, 345-346
 - dirección física, 378
 - dirección híbrida, 378
 - disco, 360-361
 - etiqueta de línea, 341
 - fallos (tasa), 336
 - impactos (tasa), 336, 346, 372, 512
 - localidad, 338
 - mapeo de dirección, 344
 - mapeo directo, 341-342
 - necesidad de, 335-338
 - nivel 1 (I1), 330-331
 - nivel 2 (I2), 330
 - organización de memoria, 335-348
 - penalización por falla, 337
 - política de colocación, 337
 - política de sustitución, 338, 347
 - rendimiento, 346-348, 468
 - tamaño de bloque, 337
 - tamaño del conjunto, 347
 - tipos de falla, 340
- Caché asociativo, 342
- Caché conjunto asociativo (*set associative*), 342-345
- Caché de dirección híbrida, 378
- Caché de dirección virtual, 377-378
- Caché de mapeo directo, 341-342
- Caché de rastreo, 351
- Caché dividida, 345
- Caché L1. *Vea* Memoria caché
- Caché L2. *Vea* Memoria caché
- Caché *snoopy*, 513-517
- Caché unificada, 345
- Caché virtualmente indexado, físicamente etiquetado, 378
- Cacheo de disco de lectura anticipada, 360
- Caída de cabeza en un disco, 355
- Calendarización, 538-539
 - algoritmo, 538-539
 - método de Brent, 546
 - tarea, 538-539
- Cámara digital, 404
- Cambio de signo en complemento a 2, 168
- Canal, 530
- Canal de bisección, 535
- Canal de eyección, 530
- Canal de inyección, 530
- Cancelación, 227, 235
- Cancelación catastrófica, 235
- Cañón de electrones, 398
- Capa metálica, 430
- Carga de entrada, 15
- Carga de fuga, 32
- Carga de salida, 501
- Carga de trabajo, 67
- Carga especulativa, 477
- Cargado, 132-133
- Cargador, 132-133
- CAS, 321-322
- CDC (Control Data Corporation), 149
- CD-ROM, 366
- Celda restador modificado (MS), 212
- Celda sumador modificado (MA), 202-203
- Cerradura de combinación, digital, 36
- Chasis, 49
- Checksum* (suma de verificación), 121
 - instrucción, 140
- Chip multiprocesador, 470, 483
- Ciclo de ejecución, 263
- Ciclo de regeneración, 321
- Ciclo de retorno, 437
- Ciclo,
 - desenrollado, 308
 - ejemplo de formación, 96
 - while* (mientras), 96
- Ciclos por instrucción. *Vea* CPI
- Cifrado de sustitución, 138
- Cilindro (en un disco), 355-356
- Circuito
 - análisis, 8
 - conmutación, 530
 - consideraciones, 15-16
 - síntesis, 8
- Circuito digital combinacional
- Circuito integrado (CI)
 - dado, 47-48
 - fabricación, 47
 - producido, 47-48
- Circuito secuencial, 21-34
- Circuito temporizador, 15-16
- CISC (computadora con conjunto de instrucciones complejas), 150
 - fórmula de Amdahl aplicada a, 150-151
- CISC, en relación con, 148-151
 - características, 149
 - impacto de rendimiento, 468
 - filosofía, 149
 - última, 151-153
- Clasificación de selección, 115-116
- CLB. *Vea* Bloque lógico configurable
- Codificación de conjunto de dígitos, 162-165
- Codificador,
 - prioridad, 13
- Codificador de prioridad, 455
- Código binario, 160
- Código de Hamming, 362
- Código unario, 160
- Coherencia basada en directorio, 519-521

- Coherencia basada en *snoop*, 513-517
- Cola, 28
- COMA, 521
- Como lógica configurable, 14-15, 30
- Compaq. *Vea* DEC/Compaq
- Comparación, 232-233
- Compilador, 53-54
- Complementador, 3
- Componente de producto, 543
- Compuerta de transmisión, 16
- Compuerta NAND, 5
- Compuerta NOR, 5
- Compuerta NOT, 4
- Compuerta OR, 4
- Compuerta XNOR, 5
- Compuerta(s),
 - arreglo de, 13-15, 31
 - carga de entrada, 15
 - red, retardo, 15
 - transmisión, 16
- Computación basada en red, 540-543
- Computación de convergencia, 235-236
- Computación de datos en paralelo, 499
- Computación distribuida, 540-543
 - impacto de rendimiento, 468
- Computadora
 - arquitecto, 39-40
 - clases, 42-44
 - empaquetado, 49-50
 - generaciones, 45-46
 - periféricos, 51-53
 - red, 53-54
 - rendimiento, 59-76
 - tecnología, 38-55
 - unidades principales, 44-45
- Computadora a bordo de automóvil, 452-453
- Computadora central, 43-44
- Computadora con conjunto de instrucciones complejas. *Vea* CISC
- Computadora con conjunto de instrucciones reducido. *Vea* RISC
- Computadora de escritorio, 42-43
- Computadora de instrucción sencilla, 151-153
- Computadora embebido, 42-43
 - en automóviles, 43, 452-453
- Computadora laptop, 42-43
- Computadora notebook, 42-43
- Computadora personal, 42-43
- Computadora personal de bolsillo, 42
- Computadora subnotebook, 42
- Computadora tablet, 42
- Cómputo de instrucciones explícitamente paralela, 474-476
- Comunicación grupal, 541
- Comunicación interprocesador, 500-501
- Comunicación persistente, 532
- Comunicación punto a punto, 529
- Comunicación(es), 51-54
 - cabecera, 434
 - grupo, 541
 - latencia, 53-54, 431
 - paso de mensaje, 528-532
 - persistente, 532
 - protocolo, 435-438
 - punto a punto, 529
- Con memoria, 21-34
- Conectable directo, 443
- Conector de nueve pines, 431
- Conector RJ-45, 432
- Conexión, 131-32
- Conexión de retardo para bifurcación, 283, 303
- Confiabilidad, 434
- Conjunto asociativo 342-345
 - dirección virtual, 377-378
 - división, 345
 - dos niveles, 337
 - etiqueta, 341
 - funcionamiento, 338-341
 - grabación directa, 338
 - grabación inversa, 338
 - indexada virtualmente, etiquetada físicamente, 378
 - política de grabación, 338, 348
 - rastreo, 351
 - snoopy*, 513-517
 - unificada, 345
- Conjunto de compartimiento, 519-521
- Conjunto de dígitos redundante, 164
- Conjunto de instrucciones
 - arquitectura. *vea* ISA
 - diseño, 147-148
 - evolución, 147-148
- Conjunto de instrucciones x86, 148
- Conjunto de trabajo, 339, 382-383
- Conmutación agujero de gusano, 530, 536
- Conmutador (conmutación) de contexto, 374, 449-461
 - cabecera baja, 460
- Conmutador de barra, 495
- Conmutador de membrana, 396
- Conmutador 2 por 2, 18, 35, 511, 531
- Conmutador mecánico, 396
- Consistencia, 147, 518-519
 - secuencial, 519
- Constante de complementación, 168-169
- Constante(s)
 - división por, 215
 - multiplicación por, 206
- Contador, 30
 - abajo, 30, 184
 - arriba, 30, 184
 - arriba/abajo, 30, 184
 - microprograma, 269-270
 - programa, 97, 185
 - síncrono, 184
- Contención de punto caliente, 512
- Conteo, 183-185
- Conteo de población, 501-502
- Control, 241-313
 - alabrado, 268
 - ciclo sencillo, 250-253
 - dependencia, 282-284
 - encauzado, 289-290
 - especulación, 476-477
 - líneas, 433
 - máquina de estado, 264-266, 272
 - microprogramado, 267-271
 - multiciclo, 258-273
 - registro, 413-415
 - riesgos, 303
 - señal, 250-253, 265-266
 - unidad, 258-276
- Control alabrado, 268
- Control basado en computadora, 407
- Control de ciclo sencillo, 250-253
- Control de estado finito para caché *snoopy*, 514
- Control encauzado, 289-290
- Controlador de semáforo, 37
- Convención de direccionamiento, 91
- Conversión A/D, 407, 441
- Conversión analógico a digital, 407, 441
- Conversión D/A, 407, 441
- Conversión digital a analógico, 407, 441
- Convertidor de código, 13
- Copiado condicional, 470
- Copiador, 280, 403
- Corrimiento
 - alineación, 227
 - aritmético, 120, 189-190
 - aritmético derecho, 120, 189-190
 - aritmético derecho variable, 120, 189-190
 - cantidad, 88
 - lógico derecho, 118
 - lógico derecho variable, 118
 - lógico izquierdo, 118
 - lógico izquierdo variable, 118
 - multibit, 190-191
 - normalización, 227
 - operación, 188-191
 - registro, 28
 - unidad, 189-190, 192
 - unidad de multietapa, 190
- Corrimiento de alineación, 227
- Corrimiento de normalización, 227-230
- Corrimiento multibit, 190-191
- Costo/rendimiento, 59-62
- CPI (ciclos por instrucción)
 - cálculo, 71-72
 - definición, 64
 - efectivo, 306
 - promedio, 70-72
 - tendencias, 308
- CPI efectivos, 286, 306
- CPU, 44
- Cray Y-MP, 511
 - algoritmo de enrutamiento, 510
 - anillo, 521-522
 - anillo de anillos, 533
 - autoenrutamiento, 510
 - bus jerárquico, 534
 - bus multinivel, 534
 - compuerta, 534
 - diámetro, 534
 - directa, 532-533
 - estación de trabajo, 540
 - etiqueta de enrutamiento, 510
 - hipercubo, 532-533
 - indirecta, 532-534
 - multietapa, 509-511
 - omega, 534
 - permutación, 511
 - procesador, 481
 - procesador a memoria, 509-513
 - procesador a procesador, 509, 513
 - toro, 500-501, 532-533
- CRC, 357
- Creación de interfaz, 440-444
- Creación de memo, 351
- Cronometrado, dos fases, 33-34
- Cronómetro, 451
- D, 22-23
 - activada por flanco, 22-23
- Dado producido, 47-48
- Datos, 103-120, 301-302
 - dependencia, 281-283, 538
 - detector de riesgos, 301-302
 - directorío de acceso, 519-521
 - distribución, 495-496, 504
 - especulación, 477
 - estado compartido, 524, 520
 - estado de sólo lectura, 514, 520
 - estado exclusivo, 514, 520
 - hacia adelante, 282, 299-301, 493
 - hacia adelante en micromips, 301
 - líneas, 433
 - movimiento en memoria, 373

- paralelismo, 499
- plantilla, 495-496
- recopilación, 440
- registro, 260-261, 413-415
- tamaños en minimips, 85, 110
- tasa, 434
- tipo, 110-113
- Datos con tamaño de byte, 110-112
- DDR, 322
 - carga de fuga, 321
 - modo de página, 322
 - tasa de datos doble, 322
- DDR-DRAM, 322
- DEC/Compaq
 - instrucciones VAX, 141, 145-146
 - procesador Alpha, 49
- Decimal
 - código, 160
 - comprimido, 175
 - conversión a binario, 166, 171
 - conversión a hexa, 166
 - división, 207-208
 - empaquetado, 162
 - multiplicación, 198-199
 - números en MiniMIPS, 87
 - punto, 169
 - sumador, 195
- Decimal codificado en binario, 162-163
- Decoder (decodificador), 12
- Decodificador BCD a siete segmentos, 9-10
- Demultiplexor, 12
- Densidad (en memoria de disco)
 - bit lineal, 354-356
 - grabación superficial, 354-356
 - pista, 354-356
- Densidad de defecto, 47-48
- Densidad de grabación (en disco), 354-356
- Densidad de grabación superficial, 354-356
- Densidad lineal de grabación, 354-356
- Departamento de Energía de Estados Unidos, 75-76
- Dependencia
 - ciclo de portadora, 492
 - control, 282-284
 - prerrequisito, 538
- Dependencia de datos, 281-282, 297-300
 - almacenamiento después de carga, 310
 - carga después almacena, 310
 - lectura después carga, 281, 298
 - lectura después del cómputo, 281, 298
 - lectura después escritura, 281
- Dependencia portada por ciclo, 492
- Desbordamiento, 135-136, 161-162
 - aritmética, 273
 - división, 209
 - punto flotante, 221, 230
 - suma, 192
- Desenrollado de un bucle, 308
- Desfase en señal de reloj, 32
- Desvío, 452
- Detección de colisión, 439
- Detector, 301-303
- Detector de contacto, 405
- Detector de luz, 404-405
- Detector de portadora, 432
- Detector de presión, 405
- Diagrama de estado, 24
- Diagrama espacio-tiempo, 279-280
- Diagrama tarea-tiempo, 279-280
- Diámetro de un disco, 354-356
- Diámetro de una red de interconexión, 534
- Diferencia de ensayo, 210
- Digital Equipment Corporation, 46
- Digital VAX, 141
 - fetching*, 263
 - formato en MiniMIPS, 86-89, 231-232
 - frecuencia, 70
 - pasos de ejecución, 87, 243-257
 - punto flotante, 230-233
 - unidad de ejecución, 246-247
 - variaciones de formato, 145-147
- Dígito de control de flujo, 535
- Dígito de protección, 234
- Dígito(s)
 - conjunto, 162-165
 - número de, 160
 - protección, 234
 - redundante, 164
- Dilema de la rapidez de la luz, 75-76
- Diodo emisor de luz, 400
- Dirección, 129
 - desplazamiento, 88
 - física, 373-376
 - línea, 433
 - predicción, 479
 - rastreo, 346
 - traducción, 325-26, 373-376
 - virtual, 373-376
- Dirección blanda, 415
- Dirección dura, 415
- Dirección física, 373-376
- Dirección virtual, 373-376
- Direccionamiento
 - desplazamiento, 88
 - física, 373-376
 - línea, 433
 - predicción, 479
 - rastreo, 346
 - traducción, 325-326, 373-376
 - virtual, 373-376
- Direccionamiento base, 97, 142-143
- Direccionamiento de actualización, 143-144
- Direccionamiento de registro indirecto, 143
- Direccionamiento directo, 98, 143
- Direccionamiento implícito, 97, 142
- Direccionamiento indexado, 143
- Direccionamiento indirecto, 532-534
- Direccionamiento inmediato, 97, 142
- Direccionamiento relativo, 143
- Direccionamiento relativo a PC, 97-98, 143
- Direccionamientoseudodirecto, 98, 143
- Directivo, ensamblador, 126-127
- Disco (memoria), 53, 353-367
 - ancho de banda I/O, 362, 423
 - área de grabación, 354
 - arreglo, 361-365. *Vea también raid*
 - cabeza lectora/grabadora, 354
 - cabezas por pista, 359
 - caché, 360-361
 - caída de cabeza, 355
 - capacidad, 354-356
 - capacidad formateada, 354
 - cilindro, 355-356
 - compacto, 53
 - densidad de grabación, 354-356
 - diámetro, 354-356
 - disco flexible, 53, 355, 365
 - eje, 354
 - latencia de acceso, 359-360
 - latencia rotacional, 355, 359
 - magnético, 53, 353-365
 - método de grabación, 356-357
 - óptico, 53, 366
 - pista, 354
 - plato, 354
 - rapidez de rotación, 354-356
 - rendimiento, 359-360
 - sector, 354
 - tabla de atributos, 356
 - tendencias tecnológicas, 385
 - tiempo de búsqueda, 355-359, 385
 - tiempo de transferencia de datos, 355
 - versátil digital, 53
 - zip, 365
- Disco cabeza por pista, 359
- Disco compacto. *Vea* Memoria de disco
- Disco duro. *Vea también* Memoria de disco
- Disco especular, 362-363
- Disco flexible, 53, 355, 365
 - sondeo, 416
- Disco magnético. *Vea* Memoria de disco
- Disco óptico escribible, 366
- Disco óptico reescribible, 366
- Disco versátil digital. *Vea* Memoria de disco
- Diseño de baja potencia, 469-470
- Diseño de circuito secuencial, 25-27
- Diseño orientado a objeto, 543
- Dispersión, 531
- Display a color Trinitron, 398-399
- Display activo, 399-400
- Display de cristal líquido, 399-400
- Display de LED, 400
- Display de marcador, 408-409
- Display monocromático, 398
- Display pasivo, 399-400
- Dispositivo
 - controlador (*controller*), 450
 - controlador (*driver*), 450
 - dirección, 415
 - manipulador, 441
 - registro de control, 413-415
 - registro de datos, 413-415
 - registro de estatuto, 413-415
 - señal *ready* (listo), 413, 418
- Dispositivo apuntador, 396-397
- Dispositivo de alojamiento de tarjetas, 49
- Dispositivo de entrada de detector, 404-407
- Dispositivo I/O de copia impresa, 400-403
- Dispositivo microelectromecánico, 367, 405
- Dispositivos periféricos, 51-53
- Distancia de Hamming, 120
- Divide (dividir)
 - instrucción, 117, 213
 - seudoinstrucción, 129
 - sin signo, 3
- División, 206-215
 - algoritmo, 206-212, 214-215
 - base alta, 211
 - binaria, 207-209
 - con signo, 209-210
 - convergencia, 212
 - decimal, 207-208
 - desbordamiento en, 209
 - ejemplo paso a paso, 207-209
 - entero, 206-210
 - fraccional, 207-209
 - iterativa, 212
 - no restauradora, 210
 - notación punto, 207
 - operandos del mismo ancho, 208-209
 - por constantes, 215
 - programada, 213-215
 - punto flotante, 229
 - recurrencia, 207
 - restauradora, 210
 - resto parcial, 207
 - sustracción de corrimiento, 206-211, 214-215

- División con signo, 209-210
- División corrimiento-restar
 - algoritmo, 197-200
 - hardware, 201-202
 - programa MiniMIPS, 205-206
 - programada, 205-206
- División de convergencia, 212
- División iterativa, 212
- División no restauradora, 210
- División programada, 213-215
- División restauradora, 210
- Divisor, 206-215
 - arreglo, 211-212
 - base alta, 211
 - hardware, 210-212
- Divisor de base alta, 211
- DMA (memoria de acceso directo), 418-421
 - controlador, 419
 - entrada desde disco, 410-421
 - método de direccionamiento, 421
 - operación, 420
 - petición de bus, 419
 - transferencia de datos, 419-421
- DMMP, 482
- DMSV, 482
- Doble palabra, 85-86, 129
- DRAM, 320-323
 - celda, 320
 - chip, 322, 345
 - longitud de ráfaga, 322
 - tendencias de capacidad, 50, 323
- DRAM de tasa de datos doble, 322
- DRAM Rambus, 322-323
- DRAM síncrona, 322
- DSL, 53
- Duodecimal (base 12), 160
- Duplicación recursiva, 504-505
- DVD-ROM, 366
- Economía de escala, 61-62
- Editor de enlaces, 132
- EEPROM, 328
- Eficiencia (de conjunto de instrucciones), 147
- EFLOPS. *Vea* FLOPS
- EIA (Electronics Industries Association), 431
- Eje, 354
- Ejecución de instrucción predicada, 475
- Ejecución fuera de orden, 306-307
- Elemento de procesamiento (PE), 502
- Elementos lógicos universales, 17
- Embarrassingly parallel*, 503
- Empaquetado, 49-50
- Empaquetado 3D, 50, 51
- Empaquetado tridimensional, 50-51
- Empujar (en pila), 107-108
- Encabezado, 535
- Encadenamiento margarita, 438-439, 443
- Encauzada, 32-33, 277-280
 - avanzada, 306-309
 - en MicroMIPS, 279
 - impacto de rendimiento, 468
 - óptima, 291-293
 - profundidad, 291
 - software, 476
- Encauzamiento de registro de estudiante, 278
- Encauzamiento dinámico, 306
- Encauzamiento en software, 476
- Encauzamiento no lineal, 295
- Encauzamiento óptima, 291-293
- ENIAC, 46
- Enrutamiento adaptativo, 537
- Enrutamiento aleatorizado, 537
- Enrutamiento basado en fuente, 537
- Enrutamiento determinista, 537
- Enrutamiento olvidado, 537
- Enrutamiento primero renglón, 537
- Enrutamiento probabilístico, 537
- Ensamblador, 53-54, 124-125
 - directiva, 126-127
 - programa, correr, 133-136
- Entero
 - con signo, 166-169
 - división, 206-210
 - representación, 110
 - sin signo, 110, 159-162
- Entrada de audio, 404
- Entrada desde teclado, 413-414
- Entrada/salida, 391-425. *Vea también* I/O
- Enunciado de caso, 100
- Enunciado *if-then-else* (si-entonces-sino), 95
- Enviar primitiva, 531
- EPC, 136
- EPIC, 474-476
- EPROM, 328
- Equilibrio, 539
- Equivalencia de expresión lógica, 7
- Error
 - acumulación, 234
 - cálculo, 233-235
 - disco, 362-364
 - punto flotante, 233-236
 - representación, 220-224, 233
- Error de cómputo, 233
- Escala de grises, 403
- Escáner, 400-401
- Escáner de alimentación de hoja, 401
- Escáner de cama plana, 401
- Escáner de exploración superior, 401
- Escáner portátil, 401
- Especulación, 476-479
- Esquema de color CMY, 403
- Esquema de color CMYK, 403
- Esquema de color RGB, 398-399
- Esquema de mapeo, 374
- Estado compartido para datos, 514, 520
- Estado consistente múltiple, 513
- Estado consistente único, 513
- Estado de alta impedancia, 5
- Estado de sólo lectura para datos, 514, 520
- Estado exclusivo para datos, 514, 520
- Estado inconsistente único, 513
- Estado inválido, 513
- Estándar
 - bus, 434-438, 441-444
 - formato de punto flotante, 172-173, 219-226
 - método de coherencia de caché, 522-523
- Estándar FireWire, 443
- Estándar Scalable Coherent Interface, 520
- Estándar SCSI, ANSI, 442
- Estimación de rendimiento, 70-72
- Ethernet, 430-432
- Etiqueta, 341
- Exaflops. *Vea* FLOPS
- Excepción, 452
 - imprecisa, 309
 - manipulador, 272
 - precisa, 309
 - punto flotante, 224-226
- Exclusión mutua, 517
- Explotación de servidor, 44
- Exponenciación, 235
- Exponente, 172-173
- Expresiones lógicas equivalentes, 7
- Extensibilidad, 147
- Extensión de signo, 110
- Extensión ISA, 470-473
 - impacto de rendimiento, 468
- Extracción de campos de una palabra, 90
- FA (sumador completo), 164, 179-180, 202-203
- Facilidad de aprendizaje/uso, 147
- Falla de arranque en frío, 340
- Falla de capacidad, 340
- Falla de colisión, 340
- Fallo de conflicto, 340
- Fallo de escritura, 342, 513-514, 520-521
- Fallo de lectura, 513-514, 520-521
- Fallo en una caché, 336
 - arranque en frío, 340
 - capacidad, 340, 374
 - colisión, 340
 - conflicto, 340, 374
 - escribir, 513-514, 520-521
 - lectura, 513-514, 520-521
 - obligatorio, 340, 374
- Fallo obligatorio, 340
- Fetch(ing)*
 - a petición, 340
 - instrucción, 263
 - política, 340, 379
- FF, 21-24. *Vea también* Flip-flop
- FF activado por flanco de bajada, 22
- FF activado por flanco de subida, 22
- Fibra óptica, 432
- FIFO. *Vea First in, first out*
- First-in, first-out* (FIFO)
 - archivo de registro, 28-29
 - buffer, 444
 - política de sustitución, 384-385
- Física, 358
- Flip-flop, 21-24
- Flip-flop D, 22-23
- Flip-flop JK, 26
- Flit, 535
- Floating-point* (punto flotante), 171-173, 219-236
- FLOPS (operaciones de punto flotante por segundo), 44, 70, 74, 468
- Flujo de caracteres I/O, 450-451
- Forma en producto de sumas, 8
- Forma en suma de productos, 8
- Formato de audio WAV, 404
- Formato de display VGA, 399
- Formato de punto flotante ANSI/IEEE, 172-173, 219-226
- Formato GIF, 404
- Formato JPEG, 404
- Formato MP3, 404
- Formato MPEG, 40
- Formato Quick Time, 404
- Formato Real Audio, 404
- Formato Shockwave Audio, 404
- Fórmula de aceleración de Amdahl, 65, 484
- Fotocelda, 404-405
- FPGA, 31-32
- Fracción inherentemente secuencial, 65
- Fracción no paralelizable, 65-66
- Fracción secuencial *f*, 65-66
- Frontera de alineación, 126
- Función de aproximación, 235
- Función logarítmica, 235
- Función trigonométrica, 235
- Fusible, 13
- Generación y verificación de paridad, 17-18
- GFLOPS. *Vea* FLOPS
- Gigaflops. *Vea* FLOPS
- GIPS. *Vea* IPS

- GMMP, 482
- GMSV, 482
- Grabación horizontal, 356-357
- Grabación NRTZ, 357
- Grabación que no regresa a cero, 357
- Grabación regreso a cero, 357
- Grabación RTZ, 357
- Grabación vertical, 356
- Grado de multiprogramación, 458
- Gran terminación, 85-86
- Grupo de estaciones de trabajo, 540
- HA (medio sumador), 179, 202-203
- Habilitación (*stroke*) de columna de dirección, 321-322
- Habilitación (*stroke*) de dirección de renglón, 321-322
- Habilitación (*stroke*), dirección de columna (CAS), 321
- Habilitación (*stroke*), dirección de renglón (RAS), 321
- Hacia adelante, 299-301
- Hallazgo máximo, 99, 115-116, 131
- Hardware
 - acelerador, 479-481
 - divisor, 210-212
 - multiplicador, 201-203
 - pila, 155
- Harvard, 345
 - conjunto de instrucciones, 81-156
 - Von Neumann, 345
- Haz de electrones, 398
- Hexadecimal (base 16), 160
 - conversión a decimal, 165
 - números en MiniMIPS, 87
 - representación de números, 163
- Hilo, 460
- Hiperhilado, 460
- IA-64, 475-477
 - procesador Itanium, 475-476
- IBM, 46, 481
 - proyecto Deep Gene, 546
- IBM System, 360-470, 141, 147
- IC, 47-48
- IDE, 444
- ILP, 473-476
- ILLIAC IV, 484, 500
- Impactar la pared de memoria, 323-325
- Impacto de rendimiento, 468
- Impacto de rendimiento del procesamiento matricial, 468
- Impactos (tasa), 336, 346, 372, 512
- Impresora, 401-403
- Impresora de inyección de tinta, 401-402
- Impresora de matriz de puntos, 401-402
- Impresora de tambor, 409
- Impresora fotográfica, 403
- Impresora láser, 402
- Incertidumbre de cronometrado, 291
- Incrementador, 184
- Incrementar, 183-185
- Incremento lineal del rendimiento, 61
- Índice, 113-115, 343
- Infiniband, 425
 - cabecera baja, 425
 - creación de redes, 406-407
 - mapeado en memoria, 413-415
 - paralelo, 518
 - socket de red, 451
- Inicialización, 133
- Iniciativa de Computación Estratégica Acelerada (ASCI), 75-76
- Iniciativa de Simulación y Computación Avanzada, 75-76
- Inmediata, 89-90
 - desbordamiento en, 192
 - inmediata sin signo, 118
 - instrucción, 89
 - sin signo, 118
 - subpalabra, 194-195
- Instrucción, 120
- Instrucción 0-*adress* (dirección 0), 142, 145
- Instrucción compleja, 139-141
- Instrucción dirección cero, 145
- Instrucción en formato I, 88
- Instrucción en formato J, 88
- Instrucción en formato R, 88
- Instrucción NOR, 89
- Instrucción OR inmediata, 90
- Instrucción OR, 89-90
- Instrucción *test-and-set* (prueba y establecimiento), 517, 524
- Instrucción(es)
 - compleja, 139-141
 - contador, 64
 - decodificador, 252, 265
 - dirección 0, 145
 - dirección 1, 145
 - dirección 1 más 1, 145
 - dirección 2, 145
 - dirección 3, 145
 - formato 80x86, 146
 - suma de verificación, 140
- Instrucciones de emulación, 122
- Instrucciones por segundo. *Vea* IPS
- Intel
- Intel Pentium, 141
 - contador de ubicación, 125
 - macro, 130-131
- Intercambio completo, 531
- Interfaz de veleta, 440
- Interfaz IEEE, 394, 442-443
- Interferencia, 15-16
- Interpolación, 236
- Interpolación lineal, 236
- Interrupción, 272, 451-460
 - confirmación, 453-454
 - habilitación, 413, 418, 453-455
 - hardware, 452
 - manipulación, 453-456
 - máscara, 453
- Interrupción enmascarada, 418
- Interrupción imprecisa, 309
- Interrupción precisa, 309
- Interrupción vectorizada, 455
- Interrupciones anidadas, 452, 456-458
 - cabecera de, 460
- Inversor controlado, 4, 6
- I/O, 395, 411-413, 422, 449-451, 502-503,
 - abstracción, 450-451
 - ancho de banda, 423
 - basado en petición, 417-418
 - benchmark* (programa de prueba) 412-413
 - bloque, 451
 - controlador, 393-395
 - copia impresa, 400
 - direccionamiento, 413-415
 - dispositivo, 51-54, 393-407
 - distribuida, 425
 - flujo de caracteres, 450-451
 - fotocopiadora, 280
 - latencia de acceso, 412, 422
 - mecánico, 396
 - membrana, 396
- memoria, 329, 345
- pérdida debida a ciclos de regeneración, 321
- pipeline*, 284
- sentencia, 100
- señales p y g, 186-187
- tamaño (en un caché), 337
- tasa de datos, 52, 412
- transferencia de datos, 412, 418-422
- I/O basado en interrupción, 417-418
 - sondeo, 417
- I/O distribuida Infiniband, 425
- I/O distribuido, 425
- I/O mapeado a memoria, 413-415
- I/O paralelo, 518
- IPS (instrucciones por segundo), 44, 49, 70-72
 - cálculo, 71-72
 - por dólar, 51
 - por watt, 469
 - puede ser engañosa, 72
- ISA, 81-156
 - cargar/almacénar, 84
- Jerarquía de memorias, 329-331
- JK, 26
- Joystick, 397
- Juego de la vida, 506
- Juego de la vida de Conway, 506
- Latch, 21-24
- Latch D, 22
- Latch SR, 21-22
- Latencia
 - acceso a disco, 359-360
 - acceso I/O, 412, 422
 - comunicación, 53-54, 431
 - encauzada, 284, 293
 - memoria, 323-324, 329-331, 384-385
 - ocultamiento, 509
 - red portadora Brent-Kung, 186
 - rotacional, 355-359
 - tolerancia, 509
- Latencia rotacional, 355-359
- Latido (en Ethernet), 432
- LCD, 399-400
- Lenguaje ensamblador, 53-54
 - programa, 123-136
- Ley asociativa, 7, 234
- Ley Cero/Uno, 7
- Ley conmutativa, 7
- Ley de Amdahl (fórmula), 65-67, 284
 - aplicada a calendarización, 539
 - aplicada a diseño, 66
 - aplicada a gestión, 66-67
 - aplicada a I/O, 412
 - generalizada, 155
- Ley de DeMorgan, 7
- Ley de identidad, 7
- Ley de Moore, 49-50
- Ley distributiva, 7
- Ley idempotente, 7
- Ley inversa, 7
- Ley Uno/Cero, 7
- Leyes del álgebra, 234
- Leyes del álgebra booleana, 7
- Librería de cinta, 367
- Limpieza a cero, 221
- Línea
 - ancho en un caché, 337, 347
 - impresora, 401
 - índice, 341
- Localidad, 338
- Localidad espacial, 338, 380

Localidad temporal, 338, 380

Lógica

- análisis de circuito, 8
- diagrama, 8
- expresión, 7
- instrucción, 89-90
- negativa, 4
- operación, 188-191
- positiva, 4
- precedencia de operador, 7
- síntesis de circuito, 8
- unidad, 191-192

Lógica de dirección siguiente, 249-250

Lógico(a)

- corrimiento. *Vea* Corrimiento
- forma en producto de sumas, 8
- forma en suma de productos, 8
- procesador, 460

Longitud de ráfaga en DRAM, 322

Longitud de vector de rendimiento medio, 497

LRU aproximado, 379-380, 384-385

MA (sumador modificado), 202-203

Macro, 130-131

Macrocelda, 31

Macroinstrucción, 130-131

Manómetro, 405

Mapa del camino de la industria de semiconductores, 49

Mapeo asociativo, 342, 379

Mapeo completamente asociativo, 342, 379

Máquina

- estado, 24-25
- estado finito, 24-25
- lenguaje, 53-54

Máquina copiadora, 280, 403

Máquina de acceso aleatorio paralela, 511-512

Máquina de estado, 24-25

- control, 264-66, 272
- receptora de monedas, 24-25

Máquina de estado finito, 24-25

Máquina de fax, 403

Máquina de Mealy, 25

Máquina de Moore, 25

MAR (registro de dirección de memoria), 153

Máscara de sombra, 398

Matriz

- acceso a subarreglo, 492-493
- almacenamiento sesgado, 496
- memoria, 325, 345

Mayor o igual, 129

Mayor que, 129

MDR (registro de datos de memoria), 153

Media geométrica, 73-74

Media palabra, 110

Medio sumador (HA), 179, 202-203

Megaglops. *Vea* FLOPS

MEM, 367, 405

Memoria, 317-389

- ancho de banda, 329, 345
- brecha de rapidez, 330
- contención de banco, 497
- de sólo lectura, 327-328
- disco, 353-367
- encauzada, 325-327
- espacio de dirección en MiniMIPS, 106-107
- flash, 328-329
- interpolada, 325-327
- jerarquía, 329-331, 384
- latencia, 323-324, 329-331, 384-385
- local, 517-519
- masiva, 353-367

matriz, 325, 345

modelo de consistencia, 518-519

no volátil, 327-329

principal, 317-331

protección, 375-376

rastreo de dirección, 346

registro de datos, 153

registro de dirección, 153

remota, 517-519

rendimiento, 324, 329

tiempo de ciclo, 336

tiempo de ciclo efectivo, 336

Memoria. *Vea* Memoria de caché

Memoria CD, 53, 366

Memoria compartida

centralizada, 508-512

distribuida, 517-524

modelo de programación, 511-512

multiprocesador, 508-524

Memoria de acceso aleatorio dinámica. *Vea* DRAM

Memoria de acceso aleatorio estática. *Vea* SRAM

Memoria de acceso amplio, 324-325

Memoria de buzón, 544

Memoria de cinta, 53, 365

Memoria de cinta magnética, 53, 365

Memoria de disco óptico, 53, 366

Memoria de sólo lectura

borrable, 328

para lógica, 30

para microprograma, 267-271

programable, 328

tecnología, 327-328

Memoria de video, 399

Memoria DVD, 366

Memoria encauzada 325-327

Memoria estilo rocola, 370

Memoria flash, 328-329

Memoria interpolada, 325-327

Memoria jerárquica, 330

movimiento de datos en, 373

Memoria masiva, 353-367

Memoria masiva casi lineal, 367

Memoria masiva fuera de línea, 367

Memoria no volátil, 327-329

Memoria principal, 317-331

Memoria virtual, 371-386

Menor o igual, 129

Menor que, 93-94, 193

Menor que inmediato, 93-94

Metacomputadora, 541-542

Método de conmutación, 530, 535

Método de grabación, 356-357

MFLOPS. *Vea* FLOPS

Microcontrolador, 42

Micrófono, 404

Microinstrucción, 267-271

formato, 268

horizontal, 271

registro, 268-269

secuenciación, 268-269

vertical, 271

Microinstrucción horizontal, 271

Microinstrucción vertical, 271

MicroMIPS, 244, 249-250, 271-273, 454-456

anidada, 452, 456-458

atención, 418, 457

contador de programa, 136

dirección destino, 88

encauzada, 309-310

encauzamiento de bifurcación, 295

multinivel, 456-458

nivel de prioridad, 452-453

no igual, 93-94

precisa, 309

predicción, 304-305

registro, 104

software, 463

solicitud, 453-454

tabla, 144

vectorizada, 455

MicroMIPS de ciclo sencillo, 253-255

tendencias, 467-470

memoria virtual, 383-386

MicroMIPS de multiciclo, 266-267

pico, 70-71

pipeline, 284-286

por watt, 469

relativo, 63

reporte de, 72-74

MicroMIPS encauzada, 286-289

Microorden, 267

Microprograma, 267-271

contador, 269-270

para MicroMIPS, 270

Microprogramable, 268

Microprogramación, 267-271

Middleware, 541

MIMD, 482

min + /min -, 173, 220-221

Minicomputadora, 44

MiniMIPS,

archivo de registro, 84

cuenta de población, 501-502

formato de instrucción, 86-89

instrucciones de punto flotante, 230-333

mix, 70, 74

multifunción, 191-193

pasos de ejecución de instrucción, 87

programa división, 214-215

programa multiplicación, 205-206

seudoinstrucciones, 129

subsistemas, 84

tabla de referencia de 30 instrucciones de punto

flotante, 233

tabla de referencia de instrucciones de 20

entradas, 98

tabla de referencia de instrucciones de 40

entradas, 119

tabla de seudoinstrucciones de 20 entradas, 129

tamaños de datos, 84

variaciones, 139-156

MiniMIPS-2D, 485-486

MIPS (mega IPS). *Vea* IPS

procesador MIPS, 49, 133

Mirroring, 362

MISD, 482-484-85

MMX (extensión multimedia), 471-473

comparación paralela, 472-473

instrucción para desempaquetar, 472

instrucciones, 471

paquete de instrucción, 472

registro, 471

MMX, comparado con, 496

multivía, 499

rendimiento, 493-494, 497-499

Modelo analítico de rendimiento, 70

Modelo de simulación de rendimiento, 70

Módem, 53

Modo, redondeo, 219-224

Motor de movimiento por pasos (gradual), 405-406

Motorola, 49

Mover

a partir de Hi, 117, 204, 213

a partir de Lo, 117, 204, 213

- punto flotante, 231-232
- seudoinstrucción, 129
- MPI, 531-532
- MS (restador modificado), 212
- MSIMD, 499-500
- Multicomputadora, 528-543
- Multicomputadora distribuida, 528-543
- Multihilo, 460-461
 - impacto de rendimiento, 468
- Multiplexor, 11-12, 16
 - en ruta de datos, 248-249, 261
 - implementación de función, 12
- Multiplicación, 197-206
 - algoritmo, 197-200, 205-206
 - algoritmo de corrimiento derecho, 197-200
 - algoritmo de corrimiento izquierdo, 216
 - base alta, 202
 - binaria, 198-200
 - complemento a 2, 199-200
 - con signo, 199-200
 - corrimiento-sumar, 197-200
 - decimal, 198-199
 - ejemplo paso a paso, 198-200
 - entero, 197-200
 - notación punto, 198
 - por constantes, 206
 - programada, 204-206
 - punto flotante, 229
 - recurrencia, 198
- Multiplicación con signo, 199-200
- Multiplicación corrimiento-sumar
 - algoritmo, 197-200
 - hardware, 201-202
 - programa MiniMIPS, 205-206
 - programada, 205-206
- Multiplicación programada, 204-206
- Multiplicador, 197-206
 - árbol, 202-203
 - árbol completo, 202-203
 - árbol parcial, 202-203
 - arreglo, 202-203
 - base alta, 202
 - hardware, 201-203
- Multiplicar-acumular, 470
- Multiplicar-sumar, 198, 470, 473
- Multiply* (multiplicar), 197-206
 - instrucción, 117, 204
 - seudoinstrucción, 129, 118, 204
- Multiprocesador, 508-524
- Multiprocesador asimétrico, 517-524
- Multiprocesador de bus compartido, 514-517
 - límite de rendimiento, 515
- Multiprocesador simétrico, 508-17
- Multiprogramación, 458
 - grado de, 458
- Multitareas, 458-59
- Mux. *Vea* Multiplexor
- NaN, 173, 224-26
 - desbordamiento, 221, 230
 - formato corto, 172-173
 - normalización, 227-230
 - número, 110, 171-173
 - operación, 226-230
 - operaciones por segundo. *Vea* FLOPS
 - raíz cuadrada, 230
 - redondeo, 219-224
 - registro, 230
 - significando, 172-173
 - subdesbordamiento, 221, 230
 - unidad en MiniMIPS, 84-85
 - valores especiales, 224-226
- NCR, 46
- Negación en complemento a 2, 168
- Niveles en jerarquía de memoria, 330
- No operación, 281
- Nombramiento de señales lógicas, 5
- Nombre global, 125
- No-op, 281
- Notación punto, 198, 207
- NOW, 540
- NUMA, 509
 - memoria compartida, 508-524
 - simétrico, 508-517
- Numeración lógica, 358
- Numerales romanos, 160
- Número completo, 159-162
 - dirigido, 166
- Número con almacenamiento de acarreo, 164
- Número con base compleja, 165
- Número con base irracional, 165
- Número con base negativa, 165
- Número de Fibonacci, 137
- Número de página física, 373-374
- Número de página virtual, 373-374
- Número de punto fijo, 169-171
- Número natural, 159-162
- Número racional, 235
- Número(s) en complemento a 2
 - base no convencional, 165
 - cambio de signo, 168
 - conversión a decimal, 168
 - conversión de base, 165-166, 170-171
 - formato de punto fijo, 169-170
 - multiplicación, 199-200
 - natural, 159-162
 - posicional, 159-162
 - precisión, 169
 - punto fijo, 169-171
 - punto flotante, 171-173
 - racional, 235
 - rango, 169
 - redundante, 164-165
 - representación, 111, 159-177
 - sistema, posicional, 159-162
 - sumador/restador, 168-169
- Oblea, 47-48
- OCR, 401
- Octal (base 8), 160
- Oculto, 1, 172-173
- Opcode (código de operación), 88
- Operador, acarreo, 185-186
- Operador, lógico, 3-6
- Operando inmediato, 88
- OR, alambrado, 6
- Ortogonalidad, 147
- OS, 55, 449-450
- Página, 371-86
- Paginación, 371-386
- Paginación a pedido, 379
- PAL, 14-15
- Palabra, 85-86, 91-92
 - significado, 113
- Palabra de instrucción muy larga, 474-476
- Pantalla táctil, 397
- Paquete, 535
 - conmutación, 530, 535
 - encabezado, 535
 - enrutamiento, 530-535
 - trailer, 535
- Par de alambres trenzados, 432
- Paralelismo a nivel de instrucción, 461, 473-476
- Paralelismo, nivel instrucción, 473-476
- Paralelismo subpalabra, 471-473
- Paridad, disco, 362-363
- Parte combinacional, 11-15
- Parte secuencial, 28-32
- Paso, 332-33, 342, 479, 492
- Paso de acceso, 332-333
- Paso de mensaje, 528-532
- PC, 147
 - sistema, 360-370, 141, 147
- PC (computadora personal), 42-43, 147
- PC (contador de programa), 97, 185
- PCI, 436-38, 442
- PCI-X, 436
- PCSpim, 135
 - calendarizado, 416-417
 - conmutador, 502-503
 - pared, 412
 - programación, 411-425
 - rendimiento, 411-413, 421-425
 - rendimiento de trabajo, 412, 422
 - tiempo de respuesta, 412, 422
 - tipos de dispositivo, 52
- PE (elemento de procesamiento), 502
- Peak, 497
 - por dólar, 51
- Pentium. *Vea* Procesador Intel Pentium
- Pérdida de significancia, 227, 235
- Petaflops. *Vea* FLOPS
- PFLOPS. *Vea* FLOPS
- Pila, 106-110
 - empujar en, 107-108
 - hardware, 155
 - máquina, 142
 - método de almacenamiento de datos, 106-108
 - puntero, 107-108
 - sacar de, 107-108
- Pipeline, 277-310
 - aceleración, 284
 - burbuja, 281-284
 - CPI efectivo, 286
 - dependencia de datos, 297-300
 - diagrama espacio-tiempo, 279-280
 - diagrama tarea-tiempo, 279-280
 - dinámica, 306
 - encadenamiento, 491-493, 498
 - excepciones, 309-310
 - interbloqueo, 300
 - latencia, 284, 293
 - límite de rendimiento, 297-310
 - pérdida, 281-84
 - producción total, 284-285
 - profundidad, 291
 - profundidad óptima, 293
 - región de arranque, 279-280
 - región de drenado, 279-280
 - registro, 288-289
 - registro de estudiante, 278
 - rendimiento, 284-286
 - riesgo de bifurcación, 302-303
 - riesgo de datos, 297-300
 - superficial, 291
 - temporización, 284-286
- PIPS (Peta IPS). *Vea* IPS
- Pista (en disco), 354
 - densidad, 354-356
- Pit (depresión en CD o DVD), 366
- PLA, 14-15
- Placenteramente paralelo, 503
- Plataforma, 541-543
- Plato, 354
- Plotter (trazador), 403

- Polinomio de aproximación, 235-236
- Política de *copyback*, 338
- Política de escritura (en caché), 338
- Política de escritura inválida, 513
- Política de grabación directa, 338
- Política de grabación inversa, 338, 513
- Política de sustitución
 - caché, 338, 347
 - global, 383
 - local, 383
 - memoria virtual, 379-382, 384-385
 - puede tener mal rendimiento, 380-381
- Política de sustitución LRU
- Política de sustitución MRU, 387
- Política de ubicación (en caché), 337, 379
- Política de uso más reciente, 387
- Poscorrimiento, 227
- Potencia computacional, 44
- Potencia de procesador de PC, 49
 - instrucciones muestra, 141
- Potenciómetro, 441
- Power PC, 141
 - predicada, 475
 - registro, 260-261
 - reordenamiento, 282
 - reutilización, 478
 - seudo, 127-130
 - sortup*, 140
 - test-and-set* (prueba y establecimiento), 517, 524
- PRAM, 511-512
- Precisión de representación, 169
- Precorrimiento, 227
- Predicción de bifurcación basada en historia, 304-305
- Predicción de bifurcación estática, 304
- Predicción de valor, 478-479
- Predicción dinámica de bifurcación, 304
- Predictor de último valor, 478
- Prefijo de suma máxima, 114-115
- Prefijos para múltiplos y fracciones de unidades, 41
- Prepaginación, 379
- Problema, 62-63, 535, 537
- Problema de día de nacimiento, 327
- Procedimiento, 103-120
 - anidado, 105
 - argumento, 103, 108-110
 - el mayor de tres enteros, 106
 - invocación, 103-106
 - llamada, 103-106
 - parámetro, 103, 108-110
 - recursivo, 121
 - resultado, 103, 108-110
 - valor absoluto, 105-106
- Procedimiento recursivo, 121
- Procesador, 44
 - tecnología, 48-51
 - tendencias de rendimiento, 50
- Procesador alfa, 49
- Procesador Am386/486, 148
- Procesador asociativo, 506
- Procesador Athlon, 148
- Procesador de red de Cisco Systems, 481
- Procesador de red Toaster2, 481
- Procesador de señal digital, 42, 469-470, 479
- Procesador Intel Pentium, 49-50
 - evolución, 148, 308-309
 - formatos de instrucción, 146
- Procesador K5/6, 148
- Procesador matricial, 482-485, 499-504
- Procesador Sun SPARC, 149
- Procesador vectorial, 493-499
 - implementación, 493-497
 - longitud de vector de rendimiento medio, 497
 - tiempo muerto, 499
- Procesador XScale, 470
- Procesamiento de transacción, 44, 413
 - programa de prueba de I/O, 413
- Procesamiento paralelo, 482-485
 - historia, 484
 - impacto de rendimiento, 468
- Procesamiento vectorial, 482-485, 490-499
 - impacto de rendimiento, 468
- Proceso, 459
- Proceso ligero, 460
- Procesos de comunicación, 538
- Producción total
 - computacional, 60-62
- Producción total de fotocopidora, 280
- Producto parcial, 198
 - acumulativo, 198
 - matriz de bit, 197
- Profundidad óptima de pipeline, 293
- Programa, 106-107, 134
 - arreglo lógico, 14-15
 - automodificable, 142
 - compilación, 54-55
 - contador, 97, 185
 - intervalo de temporización, 451
 - lógica de arreglo, 14-15
 - memoria de sólo lectura, 14-15
 - parte combinatorial, 13-15
 - parte secuencial, 30-32
- Programa ASCII, 75-76
- Programa automodificable, 142
- Programa de verificación de escritura, 137
- Programa ejecutable, 124, 132
- Programación, consciente de caché, 350
- PROM, 14-15, 328-329
- PROM borrrable (EPROM), 328
- PROM borrrable eléctricamente, 328
- Protocolo de intercambio, 436
- Proxy, 541
- Proyecto Deep Gene, 546
- Prueba de redundancia cíclica, 357
- Puerto gráfico acelerado, 444
- Punto flotante, 231-232
- Quadword, 110
- Quinario (base 5), 160
- QWERTY, 395
- RAID, 361-365
 - niveles 0-6, 363
- Raíz cuadrada
 - por convergencia, 236
 - punto flotante, 230
 - recurrencia, 236
- Rambus, 322-323
 - ciclo de refresco, 321
 - síncrona, 322
 - tendencias tecnológicas, 50, 323, 385
- Rango de representación, 169
- Rapidez de rotación, 354-356
- RAS, 321-322
- Ratón, 396-397
 - mecánico, 397
 - óptico, 397
- Rayado (en RAID), 362
- Razón SPEC, 68-69
- RCA, 46, 398
- RDRAM, 322-323
- Rebanada de tiempo, 451, 458-459
- Recepción primitiva, 531
- Reconfiguración, 480
- Reconocimiento de carácter óptico, 401
- Reconocimiento de escritura manual, 401
- Recopilación, 531
- Recurrencia
 - acarreo, 181
 - división, 207
 - multiplicación, 198
 - raíz cuadrada, 236
- Red. *Vea también* Red de interconexión
- Red ATM, 430-432
- Red de anillo, 521-522
 - cordado, 533
 - nivel dos, 533
- Red de anticipación de acarreo, 185-188
- Red de Beneš, 510
- Red de estaciones de trabajo, 540
- Red de hipercubo, 532-533
- Red de interconexión, 532-535
 - ancho de bisección, 534-535
 - anillo cordado, 533
 - mariposa, 510
- Red de permutación, 511
- Red de permutación de Waksman, 544-545
- Red directa, 532-533
- Red indirecta, 532-534
- Red mariposa, 510
- Red portadora Brent-Kung, 186-187
 - difusión, 511-512, 532
- Red procesador a memoria, 509-513
- Redes anillo de anillos, 533
- Redondeo (redondear), 219-224
 - dirigido, 172, 223-224
 - hacia +¥, 172, 223-224
 - hacia -¥, 173, 223
 - hacia 0, 173, 223
 - hacia abajo, 173, 223, 235
 - hacia adentro, 173, 223
 - hacia arriba, 127, 223-224
 - más cercano, 220-222
 - modos, 219-224
 - números ternarios, 238
 - par más cercano, 172, 221-222
- Redondeo dirigido, 172, 223-224
- Redondeo doble, 238
- Redondeo hacia abajo, 173, 223, 235
- Redondeo hacia arriba, 172, 223-224, 235
- Redondeo interno, 173, 223
- Reducción de voltaje, 469
- Reducción modular, 101
- Redundancia P + Q, 364
- Región de arranque de un encauzamiento, 279-280
- Región de drenado de un cauce, 279-280
- Register (registro)
 - aliasing*, 486-487
 - archivo, 28-29, 247
 - arquitectónico, 477-478
 - convenciones de guardado, 104-105
 - corrimento, 28
 - direccionamiento, 97, 142
 - instrucción, 260-261
 - microarquitectónico, 477-478
 - microinstrucción, 268-269
 - renombrado, 478
 - uso en MiniMIPS, 85-86
- Registro arquitectónico, 477-478
- Registro máscara, 496
- Registro microarquitectónico, 477-478
- Registros *caller-saved*, 105
- Registros, MiniMIPS, 85
- Regla de Horner, 165, 167-168
- Regreso de un procedimiento, 104

- Reloj
 - desfase, 32
 - ciclo, 32, 254, 261-262
 - frecuencia, 64, 253
 - micromips, 253, 259
 - modulación de puerta, 469
 - periodo, 32, 254, 261-262
 - política, 380
 - tasa, 64, 253
- Rendimiento, 59-76
 - bus limitado, 515
 - cacheo de disco, 361
 - definición, 62
 - estimación, 70-72
 - factor de mejoramiento, 468
 - incremento, 65-67
 - jerarquía de memoria, 384
 - medición, 67-69
 - modelado, 70-72
 - procesador de señal digital, 469
- Rendimiento de ciclo sencillo, 253-255
- Rendimiento de medición, 67-69
- Rendimiento de trabajo computacional, 60-62
- Rendimiento del modelado, 70-72
- Rendimiento multiciclo, 266-267
- Rendimiento pico, 70-71
- Rendimiento sublineal, 61
- Rendimiento superlineal, 61
- Rentabilidad reducida, 62
- Reporte de rendimiento de computadora, 72-74
- Representación
 - error, 220-224, 233
 - número, 159-173
 - punto fijo, 169-171
 - punto flotante, 171-173
- Representación megabinaria, 165
- Representación sesgada, 166
- Reserva de espacio, 126
- Reset, 21
- Resolución de referencias adelantadas, 125
- Resolución espacial, 402
- Restador con anticipación de préstamo, 194
- Resto parcial, 207
- Retardo en cola, 424
- Retícula computacional, 542-543
- Retícula de cómputo, 542-543
- Reubicación, 125, 132
- Revolución microelectrónica, 46
- Riesgo de datos, 297-300
- RISC (computadora de conjunto de instrucciones reducido)
 - fórmula de Amdahl aplicada a, 150-151
 - ventajas, 150
- Robustez, 434
- ROM. *Vea* Memoria de sólo lectura
- Rotar izquierda o derecha, 129
- RS-232, 430-431
- Ruta crítica, 254
- Ruta de datos, 241-313
 - ciclo sencillo, 247-250
 - encauzada, 277-293
 - multiciclo, 260-261
 - multiplexores en, 248-249, 261
- Ruta de datos encauzados, 277-293, 286-289
- Ruteador, 529-530
- Rutina de librería, 132
- Sacar (de pila), 107-108
- Salida a microfilm, 405
- Salida hacia unidad de despliegue, 413-415
- Salto
 - instrucción, 93-96
 - y liga de instrucción, 103-104
- SDRAM, 322
- Sector, 354
- Secuencial, 21-34
- Segmentación, 372
- Selección de chip, 318
- Selector, 11-12. *Vea también* Multiplexor
- Sensor de no contacto, 405
- Sensor de temperatura, 405
- Sentencia
 - case, 100
 - if-then-else (si-entonces-sino), 95
 - switch, 100
 - while, 96
- Sentencia while (mientras), 96
- Señal
 - control, 250-253, 265-266
 - procesador, 42, 469-470, 479
 - tiempo de propagación, 32, 433
- Señal de propagación 181, 186
- Señal enable (habilitación), 5
- Señal generadora (generar), 181, 186
- Serie de Taylor, 235-236
- Servidor, 43-44
- Sesgo, 166, 172-173
- SETI@home, 540
- Seudoinstrucción, 127-130
- Seudoinstrucción negada, 129
- Seudoinstrucción NOT, 128
- Seudoinstrucción remainder (resto), 129
- Shifter, hardware, 189-190, 192
 - basado en mux, 189-190
 - de barril, 190
 - multibit, 190-191
 - multietapa, 190
- Significando, 172-173
- SIMD, 482, 499-500
- Simulador PCSpim, 133-135
- Simulador SPIM, 133-135
- Sincronización, barrera, 517
- Sincronizador, 33, 436
- Síntesis de circuitos lógicos, 8
- SISD, 482
- Sistema, 141, 147, 360-370
- Sistema de control compartido, 499-501
- Sistema de control de ciclo cerrado, 407
- Sistema de interfaz de computadora pequeña, 442
- Sistema de legión, 543
- Sistema de software, 54-55
 - categorías, 55
- Sistema de tareas, 538
- Sistema de tiempo compartido, 542
- Sistema en un chip, 470
- Sistema Globus, 543
- Sistema numérico posicional, 159-162
- Sistema operativo (OS), 55, 449-450
- Socket de red I/O, 451
- Software de aplicación, 53-54
- Software de sistema, 53-54
- Sondeo, 416-417
- Sony Trinitron, 398-399
- Sorting (clasificación), 115-116
 - instrucción, 140
- SPEC, 68-69
- SPEC CPU 2000, 68-69
- SPECfp, 68-69
- SPECint, 68-69
- SPMD, 482, 504
- SR, 21-22, 23
- SRAM, 28, 30, 317-320
 - acceso amplio, 324-325
 - celda, 320
 - chip múltiple, 319
 - diseño de sistema, 315-389
 - estructura, 317-320
 - pared, 323-325
 - pines I/O compartidos, 319-320
 - síncrona, 318
 - tecnología, 48-50
- SRAM síncrona, 318
- SSE, 472
- SSRAM, 318
- Subdesbordamiento, 221, 230
- Subtract(ion) (resta)
 - instrucción, 89
 - sin signo, 118
- Suma con almacenamiento de acarreo, 164
- Sumador, 178-188
 - acarreo en cascada, 179-180, 182
 - almacenamiento de acarreo, 164, 202-203
 - anticipación de acarreo, 185-188
 - bits en serie, 179
 - completo, 164, 179-180
 - decimal, 195
 - híbrido, 187
 - medio, 179
 - punto flotante, 227-228
 - rápido, 185-188
 - selección de acarreo, 187-188
- Sumador completo (FA), 164, 179-180, 202-203
- Sumador con almacenamiento de acarreo (árbol), 202-203
- Sumador con anticipación de acarreo, 185-188
- Sumador con salto de acarreo, 182-183
- Sumador con selección de acarreo, 187-188
- Sumador de acarreo en cascada, 179-180, 182
- Sumador de bits en serie, 179
- Sumador híbrido, 187
- Sumador rápido, 185-188
- Supercomputadora, 42, 44
 - prueba de I/O, 412-413
 - tendencias de rendimiento, 75
- Supercomputadoras Cray, 149, 511
- Superencauzada, 306
- Superscalar, 307
 - impacto de rendimiento, 468
- Superespeculación, 478
 - basada en hardware, 477-478
 - control, 476-477
 - datos, 477
 - impacto de rendimiento, 468
- Superhilo, 460
- Tabla
 - de consulta, 236
 - de verdad, 7
 - enrutamiento, 537
 - prefijos para unidades, 41
 - referencias para instrucciones, 119, 129, 233
- Tabla de enrutamiento, 537
- Tabla de estado, 24
- Tabla de símbolos, 124-125
- Tabla de verdad, 7
- Tabla memo, 351
- Tablero de circuito impreso, 49
- Tablero PC (circuito impreso), 49
- Tamaño en caché, 347
- Tarea ligada a CPU, 62-63
- Tarea ligada a I/O, 62-63
- Tarjeta base, 49-50
- Tarjeta hija, 49

- Tarjeta madre, 49
- Tasa de datos para dispositivos I/O, 52
- Tasa de fracaso, 362
- Tasa de muestreo, 444
- Teclado, 395-396
 - ergonómico, 395
 - programación I/O, 413-414
 - sondeo, 416
- Teclado ergonómico, 395
- Teclado numérico, 395-396
- Teclado QWERTY, 395
- Tecnología
 - computadora, 38-55
 - memoria, 48-51, 317-331
 - procesador, 48-51
- Temporización de eventos, 32-34
- Temporización sensible a nivel, 33-34
- Temporizador de intervalo, programable, 451
- Tendencias de ciclo, 385
- Teorema de calendarización de Brent, 546
- Teoría de colas, 424
- Teraflops. *Vea* FLOPS
- Termina sistema, 541-543
- Ternario (base 3), 160
 - redondeo, 238
 - simétrico, 163
- TFLOPS. *Vea* FLOPS
- Tiempo de búsqueda, 355-359
- Tiempo de ciclo de memoria efectivo, 336
- Tiempo de ejecución, 62
- Tiempo de ejecución (corrido), 67
- Tiempo de establecimiento, 23, 32
- Tiempo de propagación, 32
- Tiempo de reloj de pared, 62
- Tiempo de respuesta, 62
- Tiempo de retorno, 62
- TIPS. *Vea* IPS
- TLB (*buffer* auxiliar para traducción de direcciones)
 - fallos, 377
 - relación con caché, 377-378
 - traducción de direcciones, 377
- Toro 2D, 500, 501, 532-533
- Toro, 500-501
- Touchpad* (almohadilla táctil), 397
- Trackball* (esfera móvil), 397
- Trailer, 535
- Transparencia, 147
- Transposición, 351
 - max+/max-, 161, 173, 220-221
- Trayectoria de datos de ciclo sencillo, 247-250
- Trayectoria de datos multiciclo, 260-261
- TRC, 397-399
- Triángulo en forma de aguja, 235
- Tubo de rayos catódicos, 397-399
- UART, 444
- ulp* (unidad en última posición), 169, 220
- UMA, 509
- Unicode*, 163
- Unidad aritmética/lógica. *Vea* ALU
- Unidad central de procesamiento. *Vea* CPU
- Unidad de despliegue, 399-400
 - programación I/O, 413-415
- Unidad de despliegue de color, 398-400
- Unidad de display visual, 397-400
- Unidad de ejecución e integración, 84-85
- Unidad en última posición, 169
- Unidad receptora de monedas, 26-27
- UNIVAC, 46
- URISC, 152-153
 - seudoinstrucción, 152-153
- Usada menos recientemente. *Vea* LRU
- USB, 442-443
- Valor absoluto
 - procedimiento, 105-106
 - seudoinstrucción, 128
- Valores especiales, 224-226
- Variable booleana, 7
- VDU. *Vea* Unidad de display visual
- Vecinos NEWS, 502
- Vector
 - archivo de registro, 494-495
 - operación, 491-493
 - registro de máscara, 496
- Ventajas, 483
- Videocámara, 404
- Videoprojector, 400
- Vínculo intersistema, 430-433
- Vínculo intrasistema, 429-430
- Visión abstracta de hardware, 83-86
- VLIW, 474-476
- Waksman, 543-444
- XOR
 - checksum* (suma de verificación), 121
 - compuerta, 4
 - inmediata, 90
 - instrucción, 89-90